Etude de la fiabilité des algorithmes self-convergeants face aux soft-erreurs

Greicy Costa Marques

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : Nano Électronique et Nano Technologies

Arrêté ministériel : 7 août 2006

Présentée par

Greicy Costa Marques

Thèse dirigée par **Raoul Velazco**

préparée au sein du **Laboratoire TIMA**
dans **l'École Doctorale d'Électronique, Électrotechnique, Automatique, et Traitement du Signal (EEATS)**

Étude de la fiabilité des algorithmes self-convergeants face aux soft-erreurs

Thèse soutenue publiquement le **24 octobre 2014**,
devant le jury composé de :

M. Frédéric PETROT
Professeur à l'Université de Grenoble, Président
Mme. Lirida ALVES DE BARROS NAVINER
Professeur à l'INS21 Paris, Rapporteur
M. Franck PETIT
Professeur à l'Université Pierre et Marie Curie, Rapporteur
Mme. Vania MARANGOZOVA-MARTIN
Professeur à l'Université de Grenoble, Membre
M. Raoul VELAZCO
Directeur de recherche au CNRS, Grenoble, Directeur de thèse

À mes parents

Remerciements

Ma profonde gratitude c'est en première lieu et avant tout à mon directeur de thèse, M. Raoul Velazco pour son encadrement, pour son ouverture d'esprit, sa confiance, sa patiente, sa disponibilité au cours de ces années, ses conseils et critiques qui m'ont permir de progresser et finir ce travail.

Je tiens à remercier les membres du jury:

Monsieur Frédéric Petrot, pour m'avoir fait l'honneur d'accepter de présider le jury de cette thèse.

Madame Lirida Alves de Barros Naviner et Monsieur Franck Petit, pour l'honneur qu'ils m'ont fait en acceptant d'être rapporteurs de ce travail.

Madame Vania Marangozova Martin pour sa participation à mon jury en tant qu'invitée.

Je souhaiterais remercier Madame Dominique Borrione, Directeur de Recherche au CNRS et Directeur du Laboratoire pour m'avoir accueilli au sein de TIMA ainsi que tous les administratifs du laboratoire TIMA, particulièrement Sophie Martineau pour le cours de français au début de mon arrivé à Grenoble, leur disponibilité et bonne humeur.

Je souhaite remercier Wassim Mansour et Fabrice Pancher qui m'ont fait partager leurs connaissances et compétences techniques au cours de ce travail.

Je remercie tous mes collègues qui j'ai rencontré à TIMA durant ces années, Diarga Fall, Aymen Fradi, Mariam Abdallah, André Leão, Iman Cherkawi, Michael Dimopolous, Matteo Bollo, Amaury Abrial et Yacoub Barry.

Je remercie mes amis très proches, Marisol, Silvana, Yan Xu et Etienne Rudolff qui m'ont entouré, aimé et encouragé.

Finalement, je remercie tout particulièrement mes parents Graci et Alfeu, ma soeur Jucélia, mon frère Alciney, ma belle-soeur Tânia, mon beau-frère Cido, et bien sûr mes neveux bien-aimés, Leonardo, Daniel et Juliana pour tou le soutien qu'ils m'ont apporté dans les années de ce travail.

Table des Matières

Liste des figures

Liste des tableaux

Introduction

Le progrès permanent de la technologie de fabrication des circuits intégrés conduit à des circuits potentiellement plus sensibles aux effets de la radioactivité naturelle présente dans l'environnement dans lequel ils fonctionnent [1][2]. En effet, l'impact des particules énergétiques (ions lourds, neutrons, protons, particules alpha,...) sur des zones sensibles d'un circuit intégré peuvent avoir une large palette de conséquences réunies sous le sigle SEE (Single Event Effects). Parmi les SEE, les Single-Event Upsets (SEUs) aussi appelée *bit-flips* [3][4], sont des fautes résultant en la perturbation du contenu d'une cellule mémoire.

Le très grand nombre de cellules mémoire inclues dans les circuits programmables avancés (processeurs, FPGA, systèmes sur puce, réseaux sur puce) rendent obligatoire considérer les SEU comme une source potentielle d'erreurs dans l'application finale, pour toute l'application où ces erreurs transitoires peuvent avoir des conséquences critiques. Cette situation, qui dans le passé était une préoccupation strictement pour des applications spatiales ou avioniques, doit de nos jours être considérée aussi pour les applications dédiées à opérer dans l'atmosphère terrestre, même au niveau du sol. Le phénomène SEU est considéré comme critique, car il peut conduire à des erreurs même si des techniques de tolérance aux fautes ont été mises en œuvre.

En raison de l'augmentation de la densité d'intégration des transistors de plus en plus petits, due aux progrès technologiques dans le processus de fabrication, les circuits de type processeur peuvent être dotés d'une grande puissance de calcul. Cependant, ces circuits peuvent devenir plus sensibles aux effets du rayonnement naturel. L'apparition des processeurs multicore est due initialement aux limitations physiques et technologiques des processeurs singlecores. L'une de ces limitations est la température, c'est à dire, plus de mégahertz dispose un processeur, plus ce dispositif va générer de chaleur. Ainsi, le refroidissement des processeurs *singlecore* (processeurs avec un seul core/ noyau) avec des fréquences d'horloge plus élevées est devenue une tâche de plus en plus difficile. L'un des moyens trouvés par les fabricants pour faire face à cette limitation consiste à fabriquer des processeurs à deux *cores* (*dual core*), quatre *cores* (*quad core*) ou plus (*multicore*). Néanmoins, dans le monde numérique d'aujourd'hui, des niveaux supplémentaires de sécurité, des interfaces de l'utilisateur plus sophistiquées, des bases de données plus élevées, les exigences de simulations complexes, plus d'utilisateurs en ligne, etc., demandent toujours plus de puissance de traitement. Pour que la puissance de traitement continue à marcher sur la foulée, une nouvelle architecture a été mise en place et est en vogue en ce moment : l'architecture multicore.

Dans les processeurs *multicore* avec deux ou quatre *cores* la communication entre les cores se fait par un bus commun, mais cette topologie de communication est peu évolutive, c'est-à-dire avec deux ou quatre *cores* qui partagent le même bus, la performance n'est pas grandement affectée, cependant, avec un plus grand nombre *cores*) le contrôle de l'accès au bus est plus complexe ainsi que l'accès à la mémoire sature vite. Le *crossbar* est une topologie qui permet de surmonter certaines des limitations du bus. Le commutateur ou *matrice crossbar* est une alternative non bloquante d'interconnexion. Tous les éléments peuvent être reliés entre eux de manière dynamique, mais cette caractéristique est originaire de la grande disponibilité du matériel, ce qui se traduit par un coût élevé du système pour un grand nombre de cores. Cependant, il n'est pas évolutif, il présente des points de contrôle excessifs et a un coût élevé. Mais comme la tendance est d'augmenter le nombre de *cores* dans un processeur et de maintenir ainsi une puissance de traitement élevée, l'idée d'une communication segmentée divisée a été explorée [5]. Ce concept se traduit par un réseau de routage composé par des liens de communication de données et des routeurs qui sont mis en œuvre sur une puce [6]. La proposition de base de ces réseaux sur puce, appelés NoC (*Network on Chip*) est : le développement d'interconnexions sur puce doit suivre les mêmes principes qui sont appliqués dans la conception d'un réseau de communication à un niveau macroscopique, qui démontrent l'évolutivité durable, l'amélioration exponentielle des performances et une fiabilité et robustesse exceptionnelles.

Vu l'avènement des processeurs *multicore*, la tendance pour établir une communication entre les *cores* est l'utilisation des réseaux sur puce. Il est donc souhaitable que les communications entre les *cores* soient fiables, c'est à dire que certaines tâches ne peuvent pas permettre même une faible probabilité d'erreur. Dans ce contexte les algorithmes dits *auto-convergeants* peuvent être nécessaires. Des exemples de processeurs multicore en utilisant un réseau sur puce pour aider à atténuer le problème de communication entre les *cores* peuvent être trouvés dans la référence [92] . Ce travail considère un processeur 48-cores Intel SCC, alors que dans les références [93] et [94] sont considérés respectivement un processeur 64-core Tile64 et un processeur expérimental avec 80-cores. Dans ces travaux le réseau sur puce décentralise la communication à travers de la puce en utilisant un protocole réseau léger, résultant en un haut débit et en une solution évolutive. En effet ces algortithmes convergent, automatiquement et indépendamment de l'état initial du système, à un ensemble d'états qui répondent à la spécification du problème. Due à la propriété d'autostabilisation, les algorithmes d'auto-convergence sont utilisés, par exemple, pour grouper les réseaux à grande échelle (*Cluster*) afin d'assurer l'extensibilité, ainsi que dans les réseaux de capteurs, etc.

L'algorithme d'auto-convergence est considéré comme étant intrinsèquement tolérant aux fautes transitoires, et a comme propriété importante *l'autostabilisation* qui permet au système distribué tolérer des défaillances transitoires, en particulier des défaillances qui ne modifient pas le code exécuté par n'importe quel nœud. Après une défaillance ou une série de défaillances qui corrompent les données, le système atteint une configuration arbitraire (c'est à dire une configuration

où les variables ne sont pas initialisées), dans un temps fini, et finira pour récupérer un comportement correct si aucune autre faute se produit. Dans cette thèse des techniques de tolérance aux fautes ont été implémentées au niveau logiciel dans l'algorithme d'auto-convergence avec le but d'améliorer sa robustesse face aux SEUs. Plusieurs campagnes d'injection de fautes ont été effectuées pour évaluer les éventuelles faiblesses de l'algorithme ainsi que pour vérifier la fonctionnalité de l'algorithme d'auto-convergence modifié.

Une contribution significative de ces travaux est liée au fait de considérer les SEUs comme une menace pour les algorithmes auto-convergeants. Ces algorithmes peuvent être exécutés par un processeur dans une architecture traditionnelle (c'est à dire, avec processeur, mémoire SRAM ...), par un processeur implémenté dans un FPGA, par un circuit dédié, ASIC (Application Specific Integrated Circuit), ou par un circuit au sein d'un dispositif complexe, comme par exemple un processeur *multicore*. Les algorithmes auto-convergeants quand exécutés par un processeur par exemple, sont exposés aux fautes transitoires (particulièrement aux SEUs) provoquées par exemple par les radiations présentes dans l'environnement, ceci en raison de la sensibilité aux SEUs de ses cellules mémoire (registres, flip flops, caches d'instructions et de données, …). A titre d'exemple, si les SEUs affectent des données d'entrée ou de sortie de l'algorithme, l'algorithme convergera mais pas au bon résultat, c'est à dire ces données ne peuvent pas être corrigées par l'algorithme ce qui empêchera sa convergence vers une valeur correcte.

L'objectif principal de cette thèse est l'étude de la robustesse/sensibilité face aux SEU's d'un algorithme d'auto-convergence. Cette étude sur l'auto-convergence et sa robustesse/sensibilité aux effets des radiations naturelle est la première dans ce domaine et pourra avoir un impact important, comme dit précédement,vu la conjoncture de miniaturisation qui permettra bientôt de disposer de circuits avec des centaines à des milliers de cœurs. Avec la possibilité d'intégrer des centaines à de milliers de cœurs de traitement sur une seule puce [5], il faudra faire les cœurs communiquer de manière efficace et robuste. Dans ce contexte les algorithmes dits auto-convergeants peuvent être utiles afin que la communication entre ces cœurs soit fiable et sans intervention extérieure.

Cette thèse s'articule autour de quatre chapitres. Le premier chapitre est consacré à la présentation des généralités des radiations : les types de radiation, les radiations qui menacent potentiellement la fiabilité des dispositifs électroniques et les acteurs qui sont impliqués dans la formation de l'environnement radiatif spatial et atmosphérique. Ce chapitre se termine par la présentation des effets des radiations sur les circuits intégrés.

Dans le deuxième chapitre sont présentés les systèmes distribués, l'autostabilisation ainsi que un benchmark de l'algorithme d'auto-convergence. Ce chapitre discute d'abord les motivations pour les systèmes distribués, les généralités de ces systèmes ainsi que leurs caractéristiques propres. Puis sont présentés les critères pour la classification des fautes dans les systèmes distribués et l'approche

pour la tolérance aux fautes ainsi que les défis de la mise en œuvre de ces systèmes et leur modélisation. L'approche d'autostabilisation, les propriétés qui doivent être satisfaites pour que le système soit autostabilisant et l'autostabilisation dans la pratique sont présentés. Enfin, dans la dernière partie nous présentons l'algorithme d'auto-convergence, appelé PCCAS (Plus Courts Chemins Auto Stabilisants), son fonctionnement et ses problèmes potentiels.

Dans le troisième chapitre est présentée une étude par injection de fautes de la robustesse de l'algorithme d'auto-convergence étudié. Cet algorithme a été exécuté par un processeur LEON3 implémenté dans un FPGA embarqué dans une plateforme de test spécifique, le testeur ASTERICS développé à TIMA dans le cadre des recherches précédentes. Dans ASTERISC est implémentée une méthode qui permet l'injection de fautes de type SEU en utilisant les interruptions asynchrones, approche CEU (Code Emulated Upset), développée à TIMA dans le cadre des recherches antérieures. Les campagnes préliminaires d'injection de fautes ont mis en évidence une certaine sensibilité aux SEU's de l'algorithme étudié. Pour y faire face des modifications du logiciel ont été effectuées et des techniques de tolérance aux fautes ont été implémentées au niveau logiciel dans le programme implémentant l'algorithme d'auto-convergence. Des campagnes d'injection de fautes ont été effectués pour mettre en évidence la robustesse face aux SEUs de l'algorithme modifié et ses potentiels « Tallons d'Achille ». À la fin de ce chapitre est donnée une version améliorée de l'algorithme d'auto-convergence incluant différentes stratégies (modifications logicielles et l'implémentation d'un TMR implémenté dans un processeur LEON3 trois *cores*) mettant en évidence sa robustesse face aux SEUs.

Le quatrième chapitre décrit l'implémentation d'un circuit auto-convergeant dans l'un des FPGA de la plateforme de test ASTERICS ainsi que la méthode d'injection de fautes NETFI (NETlist Fault Injection) [7], développée à TIMA dans le cadre des recherches antérieures, méthode qui sera utilisée pour obtenir des résultats sur la robustesse intrinsèque face aux SEUs de l'implémentation d'un circuit auto-convergeant. Dans cette étape l'injection de fautes (NETFI) est faite au niveau du modèle RTL du circuit. Les résultats issus des campagnes d'injection de fautes sont présentés et comparés à ceux obtenus avec le processeur LEON3 exécutant la version logicielle de l'algorithme étudié.

Le dernier chapitre présentera les conclusions qui feront la synthèse de ces recherches ainsi que les futures perspectives de ces travaux.

Chapitre 1. L'environnement radiatif et ses effets sur les circuits intégrés

Dans ce chapitre sont introduites les généralités sur les radiations ainsi que les environnements radiatifs spatial et atmosphérique, et leurs effets sur les circuits intégrés.

La première section établit les généralités sur les radiations : les types de radiation, les radiations qui menacent potentiellement la fiabilité des dispositifs électroniques, et les « acteurs » qui sont impliqués dans la formation de l'environnement radiatif. La deuxième section discute les environnements radiatifs spatial et atmosphérique, dans lequel opèrent les composants électroniques. La troisième section introduit les effets des radiations sur les circuits intégrés : les mécanismes d'interaction, les effets de dose, et les événements singuliers.

1.1 Les généralités sur les radiations

Les dispositifs microélectroniques et les circuits intégrés peuvent être exposés à une vaste gamme d'environnements radiatifs. Les types de particules, leurs énergies, leurs flux et fluences (ou dose totale) peuvent varier considérablement entre les différents environnements radiatifs auxquels les dispositifs électroniques peuvent être exposés. Ces différences peuvent entraîner des variations importantes de la dégradation induite par radiation.

Parmi les types de radiation présents dans la nature, les rayons cosmiques qui atteignent la surface terrestre sont une menace potentielle à la fiabilité des dispositifs électroniques. La Terre offre une bonne protection à travers de sa **magnétosphère** et de **l'atmosphère**, qui pour les particules chargées correspond à un mur de béton épais [8]. La magnétosphère terrestre est une cavité naturelle dans le milieu spatial dans laquelle la Terre est relativement protégée des influences extérieures. Cette cavité est aplatie côté Soleil et étirée côté nuit. La Figure 1.1 montre l'interaction entre le vent solaire et la magnétosphère terrestre. L'atmosphère terrestre désigne l'enveloppe gazeuse entourant la Terre. L'air sec se compose de 78% d'azote, 21% d'oxygène, le reste étant constitué de gaz rares comme l'argon, de gaz carbonique, de vapeur d'eau et de traces d'autres constituants, sans oublier des particules en suspension. Les particules en suspension sont des fines particules solides portées par l'eau ou solides et/ou liquides portées par l'air. Ces particules en suspension (poussières, sel, sable,

pollens, graines...) se déposent très lentement selon les déplacements des masses d'air et sont généralement amenées au sol par les précipitations.

Les satellites en orbite terrestre basse (*low earth orbit - LEO*), c'est-à-dire à une altitude entre 500 et 2000 km d'altitude, n'ont pas la protection de l'atmosphère, mais sont protégés par la magnétosphère. Au contraire, les satellites en orbite géostationnaire (*geostationary orbit - GEO*), située à 35 784 kilomètres d'altitude au-dessus de l'équateur de la Terre, utilisés principalement pour les communications intercontinentales, n'ont pas la protection de la magnétosphère, et sont soumis à toutes sortes de radiations issues de l'espace (le soleil aide à protéger les satellites des radiations issues de l'environnement interstellaire mais il est lui-même un émetteur majeur de protons énergétiques par le **vent solaire**).

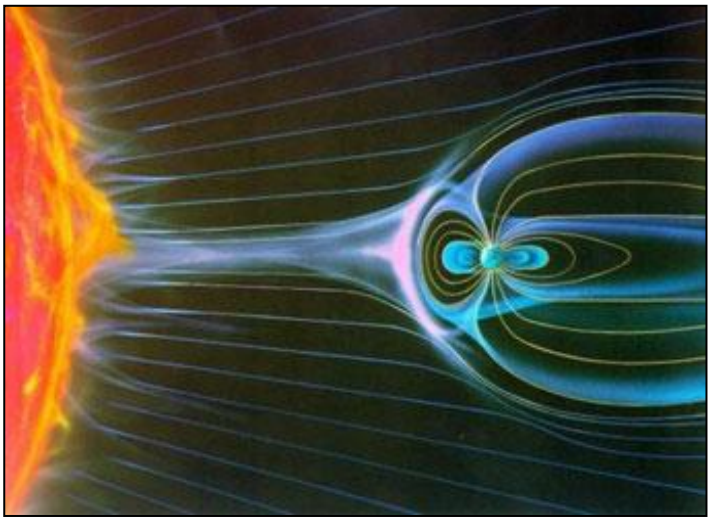


Figure 1.1- Vue d'artiste de l'interaction entre le vent solaire et la magnétosphère terrestre

Plusieurs études ont été faites sur les radiations et leurs conséquences sur les circuits intégrés. Les premières études datent de la fin des années 70. Dans une première publication qui date de 1975 [9], est faite une étude sur les anomalies provoquées par les rayons cosmiques galactiques dans les composants logiques embarqués à bord d'un satellite. En 1979 [10], une importante attention des communautés scientifiques est portée sur les techniques pour observer les perturbations dues aux interactions entre les neutrons et le silicium aux altitudes avioniques ainsi qu'au niveau du sol.

Parmi les composants électroniques vulnérables aux radiations, nous pouvons citer les mémoires. Les mémoires sont loin d'être les seuls composants électroniques vulnérables aux radiations, bien qu'il est clair que les cellules mémoires étant présentes dans la plupart de composants digitaux constituent une cible significative pour les radiations. Si l'on considère par exemple qu'une mémoire possède le code du programme exécuté par une application et qu'une particule chargée change le contenu d'un bit dans une ou plusieurs cellules mémoires, dépendant de la nature de l'information contenue dans les cellules perturbées et de l'instant d'occurrence ce changement pourrait

provoquer par exemple le blocage du logiciel embarqué. Cependant, les erreurs dans des mémoires peuvent être détectées et/ou corrigées par des techniques bien connues comme: la parité, les codes de Hamming, le code de R-S (Reed - Solomon), etc. [11][12][13][14][15].

La Terre est protégée par l'atmosphère. Celle-ci arrête la plus grande partie des radiations issues de l'espace, mais les particules qui franchissent l'atmosphère présentent un niveau élevé de dangerosité pour la fiabilité des circuits intégrés.

Au niveau du sol, même avec la protection offerte par l'atmosphère terrestre, des particules énergétiques (neutrons par exemple) peuvent atteindre et perturber le fonctionnement des composants électroniques. Les progrès permanents des technologies de fabrication des circuits intégrés conduisent à des circuits potentiellement plus sensibles aux effets de la radiation naturelle présente dans l'environnement dans lequel ils fonctionnent. La prise en compte des dommages dues aux radiations est donc un point crucial pour assurer le bon fonctionnent des systèmes qui évoluent dans l'espace, dans l'atmosphère terrestre et aussi ceux qui opèrent au niveau du sol et dont les pannes peuvent avoir des conséquences critiques (transport, équipements pour la médecine, …).

1.2 Environnements Radiatifs

Cette section présente l'environnement radiatif naturel de l'espace et l'environnement radiatif atmosphérique, dans lequel opèrent les composants électroniques.

1.2.1 L'environnement radiatif spatial

Dans l'environnement spatial, les perturbations induites par les rayonnements ionisants sont responsables de disfonctionnements dans les circuits intégrés. Les perturbations dans le fonctionnement des circuits intégrés ont des dépendances avec les caractéristiques des particules (énergie et flux) ainsi que de leur probabilité d'apparition. La fiabilité des circuits intégrés dans l'environnement spatial est plus complexe et plus contraignante. La large variété des particules et la quasi-inaccessibilité des systèmes au cours de leurs missions rendent l'évaluation de la fiabilité une étape à la fois délicate et cruciale.

L'environnement radiatif spatial est constitué de particules couvrant un spectre très large qui s'explique par l'origine diverse de ces sources ionisantes. Les principaux composants de l'environnement radiatif spatial sont classés, suivant leur origine, en quatre catégories : le vent et les éruptions solaires, le rayonnement cosmique et les ceintures de radiations [16]. Dans les sections suivantes sont donnés des détails sur chacune de ces catégories.

1.2.1.1 Le vent solaire

Le vent solaire est un flux de plasma constitué essentiellement d'ions (tels que l'oxygène ou le carbone), d'électrons, et d'hélium (7 à 8%) qui sont éjectés de la couronne solaire. La couronne est la partie de l'atmosphère du soleil au-delà de la chromosphère. La chromosphère est la basse atmosphère du soleil. C'est une fine couche rose de gaz, transparente pour la lumière visible, située entre la photosphère et la couronne solaire. La photosphère est la couche de gaz extérieure du soleil, qui en donne l'image visible et qui fournit la majeure partie du rayonnement lumineux et calorique. La chromosphère est néanmoins observable lors d'une éclipse totale du soleil à l'aide d'un coronographe. Un moyen d'étudier la chromosphère est observer le soleil dans une longueur d'onde correspondant à une raie (656,3 nm) de l'hydrogène appelée H_α. Dans cette longueur d'onde les atomes d'hydrogène de la chromosphère absorbent la lumière de la photosphère et la réémettent vers l'extérieur. La densité du plasma correspond à $10^{12}cm^{-3}$ au niveau du soleil et tombe à environ 10 particules/cm^3 au niveau de l'orbite terrestre. Une coupe avec les trois couches : la couronne solaire, la chromosphère et la photosphère est montrée dans la Figure 1.2.

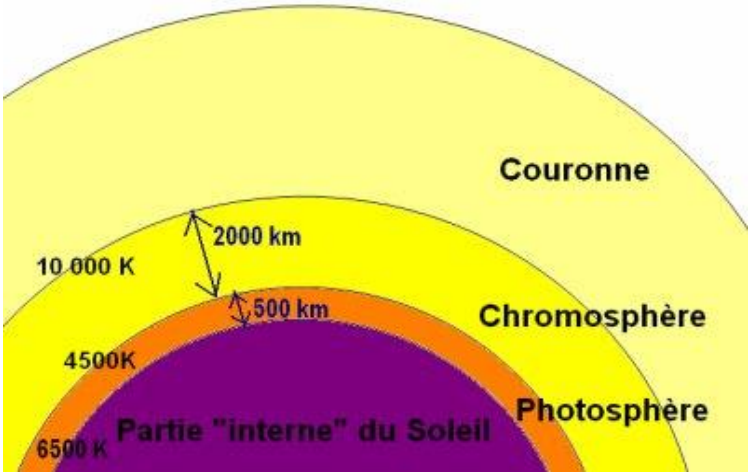


Figure 1.2 - Coupe simplifiée avec les trois couches du soleil: la couronne, la chromosphére et la photosphère

La Figure 1.3 représente au milieu le soleil avec tout autour la couronne solaire. Le graphique montre la vitesse du vent solaire mesurée par le satellite Ulysse : le vent solaire rapide est expulsé des régions autour des pôles (Nord et Sud) du soleil à la vitesse de 800 km/s, alors que le vent solaire lent afflue de la zone équatoriale à 350 km/s. Le vent solaire est immergé dans un champ magnétique. Le graphique est coloré en rose quand la polarité du champ magnétique solaire pointe dans la direction opposée au soleil (au cours du cycle solaire actuel, il s'agit de la direction du champ magnétique au-dessus du pôle Nord du soleil) et est coloré en bleu quand la polarité du champ magnétique solaire pointe vers le soleil (la direction du champ magnétique au-dessus du pôle Sud).

1.2.1.2 Les éruptions solaires

Une éruption solaire est un événement primordial de l'activité du soleil. La variation du nombre d'éruptions solaires permet de définir un cycle solaire (cycle magnétique) d'une période moyenne de 11 ans (environ 4 ans de faible activité et 7 ans de forte activité) ponctuée par des événements mineurs, majeurs et exceptionnels. La Figure 1.4 illustre sous forme graphique les cycles solaires depuis l'année 1700 jusqu'à l'année 1997.

Le système de numérotation des cycles solaires a été mis au point au milieu du XIXe siècle, plus particulièrement en 1848 par Rudolf Wolf de l'observatoire de Zurich. L'éruption se produit périodiquement à la surface de la photosphère. La photosphère est la couche de gaz de quelques centaines de kilomètres d'épaisseur et de température moyenne d'environ 5800K (Kelvin), elle constitue la surface visible des étoiles, en particulier du soleil, où se forment la plupart des raies spectrales depuis l'ultra-violet jusqu'à infrarouge. Elle projette au travers de la chromosphère (couche irrégulière entourant la photosphère d'une épaisseur avoisinant les 10000 km) des jets de matière ionisée qui se perdent dans la couronne à des centaines de milliers de kilomètres d'altitude.

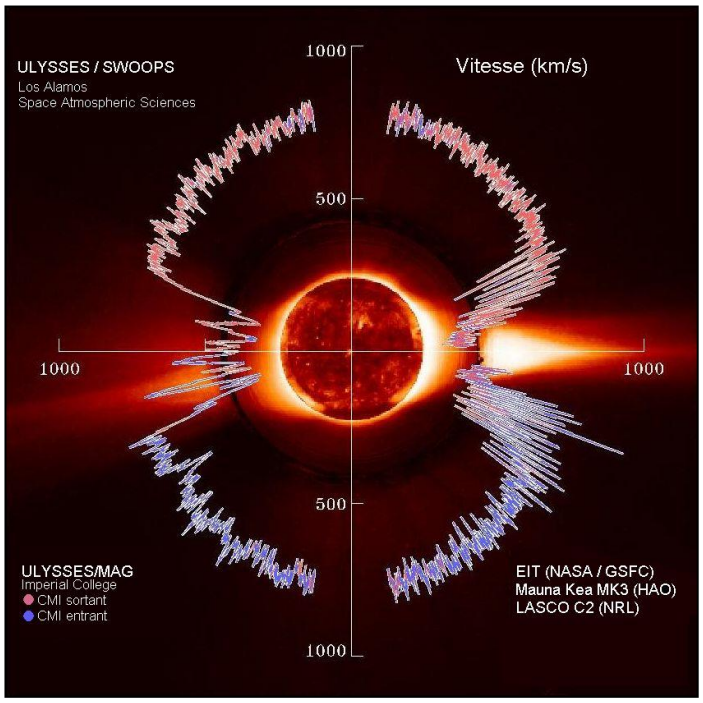


Figure 1.3 – La couronne solaire. Mesure du vent solaire par le satellite Ulysse

Durant les éruptions solaires, d'importants flux de protons énergétiques sont produits et atteignent la Terre. De tels évènements sont imprévisibles tant du point de vue du moment auquel ils apparaissent, que de leur magnitude, leur durée ou leur composition. Le champ magnétique terrestre

protège de ces particules une région de l'espace proche de la Terre (bouclier géomagnétique), mais elles atteignent facilement les régions polaires et les hautes altitudes telles que les orbites géostationnaires.

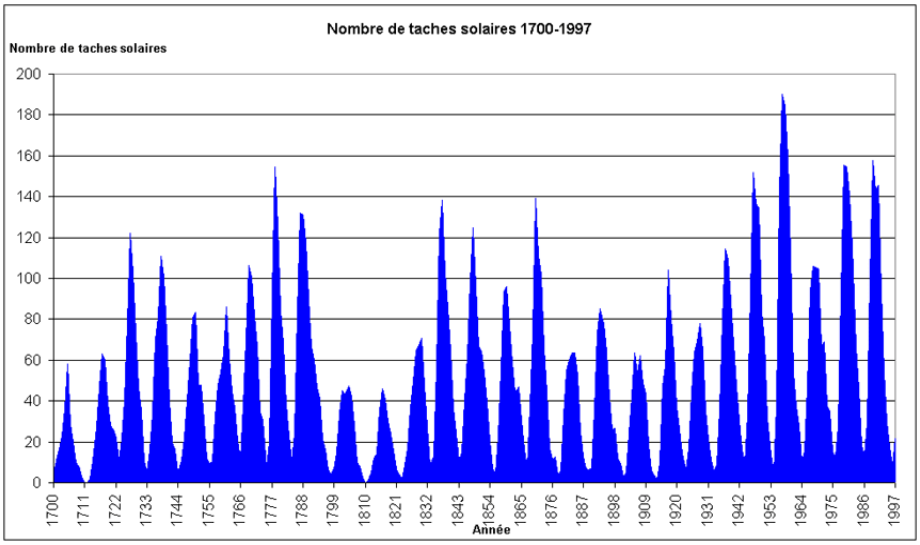


Figure 1.4 – Représentation de cycles solaires depuis 1700 [17]

Les éruptions solaires suivent trois stades, chacun d'eux pouvant durer de quelques secondes à quelques heures selon l'intensité de l'éruption. Durant le stade précurseur, l'énergie commence à être libérée sous la forme de rayons X. Puis les particules, électrons, protons et ions lourds, accélèrent jusqu'à approcher la vitesse de la lumière lors du stade impulsif. Le plasma se réchauffe rapidement, passant de quelques 10 MK à 100 MK. Une éruption donne non seulement un flash de lumière visible, mais émet également des radiations dans le reste du spectre électromagnétique : rayons gamma, ondes radio et rayons X. Le stade final est le déclin, pendant lequel les rayons X mous sont une fois de plus les seules émissions détectées.

La première éruption solaire observée date du 1er septembre 1859. Elle fut faite par l'astronome britannique Richard Carrington, lorsqu'il constata l'apparition d'une tache très lumineuse à la surface du soleil qui perdurait pendant 5 minutes. La Figure 1.5 montre un croquis du groupe de taches solaires à l'origine de l'éruption solaire, dessiné par Richard Carrington. Une tache solaire est une région sur la surface du soleil (photosphère) qui apparaît noire à l'observation visuelle. Cette région est marquée par une température inférieure à son environnement et à une intense activité magnétique. Le champ magnétique de cette région remonte la zone convective, son intensité agit comme une muraille et empêche le plasma de remonter. En raison de ce fait, le plasma ne peut pas sortir, ce qui donne à la place des taches sombres, voir Figure 1.6. Dans la Figure 1.5 les quatre zones étiquetées de A à D correspondant aux lieux où apparurent les flashes (taches blanches) de l'éruption et dans la Figure 1.7 est montrée une éruption solaire prise par le satellite TRACE de la NASA.

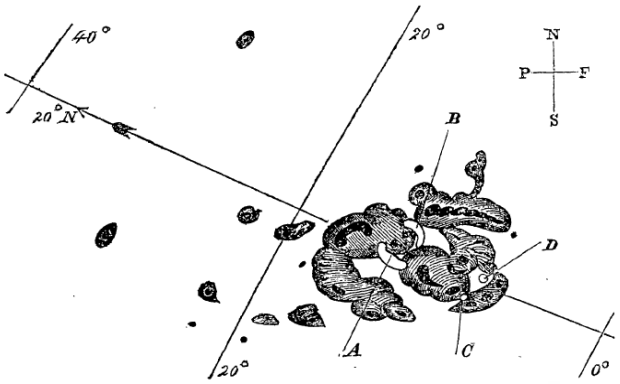


Figure 1.5 – Ce dessin du 1er septembre 1859 est le premier enregistrement des éruptions solaires

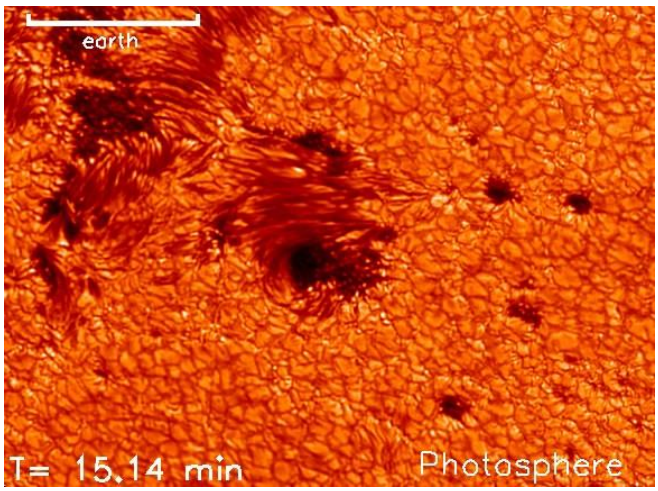


Figure 1.6 - Taches solaires

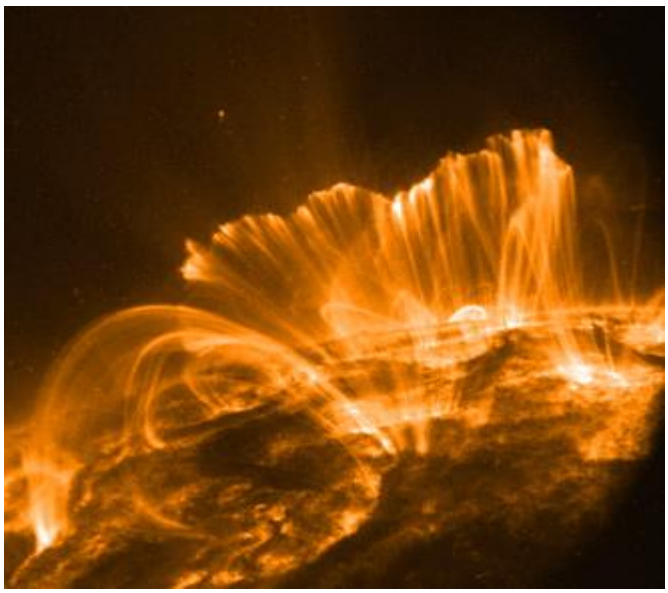


Figure 1.7 – Une éruption solaire prise par le satellite TRACE de la NASA

1.2.1.3 Le rayonnement cosmique

Le rayonnement cosmique est le flux de noyaux atomiques et de particules de haute énergie qui circulent dans le vide interstellaire. Il a été découvert par Victor Hess en 1922, dont cette découverte lui valut un prix Nobel, grâce à des mesures effectuées à partir de ballons-sondes. L'origine de ce rayonnement cosmique est mal connue, mais on sait toutefois qu'une partie, correspondant aux ions les plus énergétiques, est extragalactique et que l'autre partie est composée entièrement de rayonnement galactique. Les flux de ces particules sont faibles mais, parce qu'elles incluent des ions lourds et énergétiques, elles induisent une très forte ionisation quand elles traversent la matière : certains ions atteignent l'énergie de 10^{11} GeV (mais les mécanismes d'accélération correspondants ne sont pas toujours très bien compris).

Il est principalement constitué de protons (85%) et de noyaux d'Hélium (14%) [18]. Moins de 1% du spectre de rayonnement cosmique est composé d'ions lourds de très grande énergie (énergie > 1 MeV). La Figure 1.8 [19][20] montre l'abondance relative des ions cosmiques.

Ces particules ne sont toutefois pas les principaux responsables des dégradations dans les composants électroniques. Cependant, il est difficile de s'en protéger et donc sont l'objet d'un danger significatif. Ces particules peuvent donner lieu à des évènements singuliers (SEU) dans les circuits fortement intégrés [21].

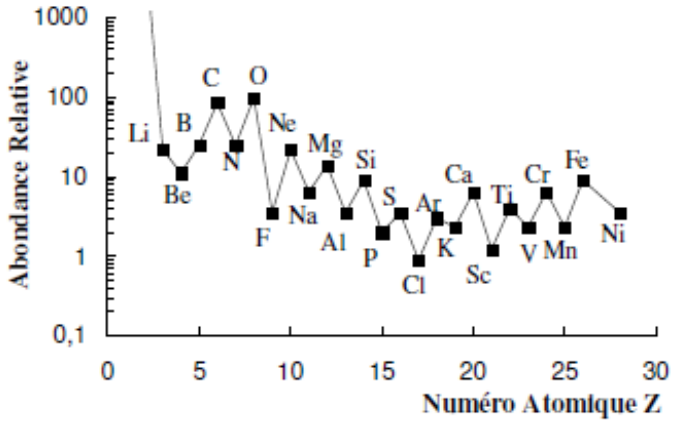


Figure 1.8 – Abondance relative d'ions cosmiques

1.2.1.4 Les ceintures de radiations ou ceintures de Van Allen

Le champ magnétique terrestre crée une cavité géomagnétique connue sous le nom de magnétosphère [19]. Les lignes de champ magnétique peuvent piéger les particules chargées de faible énergie. Ces particules piégées sont essentiellement des électrons et des protons, mais quelques ions lourds sont aussi piégés. Les zones de particules piégées sont appelées ceintures de radiation ou ceintures de Van Allen et ont une forme toroïdale nichée à l'intérieur de la magnétosphère [16]. Des

électrons piégés avec des énergies comprises entre quelques dizaines d'eV et 10 MeV et des protons piégés avec des énergies comprises entre 100 KeV et plusieurs centaines de MeV constituent les ceintures de radiations. Les particules piégées effectuent un mouvement spiral autour des lignes de champ magnétique et sont reflétées dans les deux sens entre les pôles où les champs sont confinés. Ce mouvement est illustré dans la Figure 1.9 [19].

Comme les particules chargées tournent autour des lignes de champ magnétique, elles dérivent également autour de la Terre : les électrons en direction de l'Est et les protons en direction de l'Ouest. Il existe principalement trois ceintures de radiations. Deux ceintures sont formées d'électrons à haute énergie et se situent à 9 000km et 30 000 km d'altitude. La troisième ceinture est constituée principalement de protons à haute énergie et se trouve à 12 000 km d'altitude.

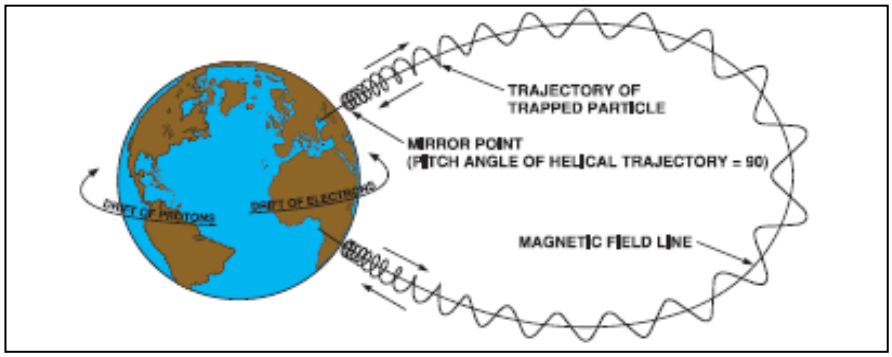


Figure 1.9 – Mouvement des particules piégées dans magnétosphère terrestre

Au-dessous de l'océan Atlantique centré au large des côtes d'Amérique du Sud, la sphère géomagnétique plonge vers la Terre, provoquant une région du flux de protons à des altitudes relativement basses. Cette région est appelée l'anomalie de l'Atlantique du Sud (SAA : South Atlantic Anomaly). Elle existe en conséquence du décalage entre l'axe de rotation géographique de la Terre et son axe magnétique. Dans cette région, le flux de protons avec des énergies supérieures à 30 MeV peut être jusqu'à 104 fois plus élevé que celui dans des altitudes comparables dans d'autres régions de la Terre. En altitudes plus élevées la sphère magnétique est plus uniforme et l'anomalie de l'Atlantique Sud disparaît [22][23]. La Figure 1.10 montre les cartes avec les courbes de protons à 500 km, 1000 km et 3 000 km d'altitude, indiquant clairement la localisation de l'SAA à baisse altitude (à 500 km).

À cause des particules du dipôle magnétique terrestre, les ceintures de radiations sont plus basses au niveau du Brésil, voir Figure 1.10. Un satellite en orbite basse (LEO – Low Earth Orbit) verra une asymétrie dans l'exposition aux radiations qu'il recevra.

1.2.2 L'environnement radiatif atmosphérique

La Terre et son environnement sont protégés des radiations cosmiques par son champ magnétique et son atmosphère. Ceux-ci fonctionnent comme une barrière naturelle arrêtant la plus

grande partie des radiations issues de l'espace. Cependant, avec l'avancement des technologies de fabrication des circuits intégrés, les particules qui franchissent cette barrière présentent un niveau de dangerosité qui ne cesse de croître vis-à-vis de la fiabilité des systèmes électroniques.

L'environnement radiatif atmosphérique résulte de l'interaction des rayonnements cosmiques avec les atomes constituant les molécules de l'atmosphère (entre autres : 78% d'azote et 21% d'oxygène) [24]. Cette interaction crée des particules secondaires qui peuvent prendre la forme d'une cascade comme montré dans la Figure 1.11. Ces particules peuvent être des protons (p), électrons (e^-), neutrons (n), ions lourds, muons (μ) et pions (π). Toutes ces particules peuvent potentiellement interagir avec la matière. Les énergies de ces particules sont comprises entre 0 et 10^{11} eV.

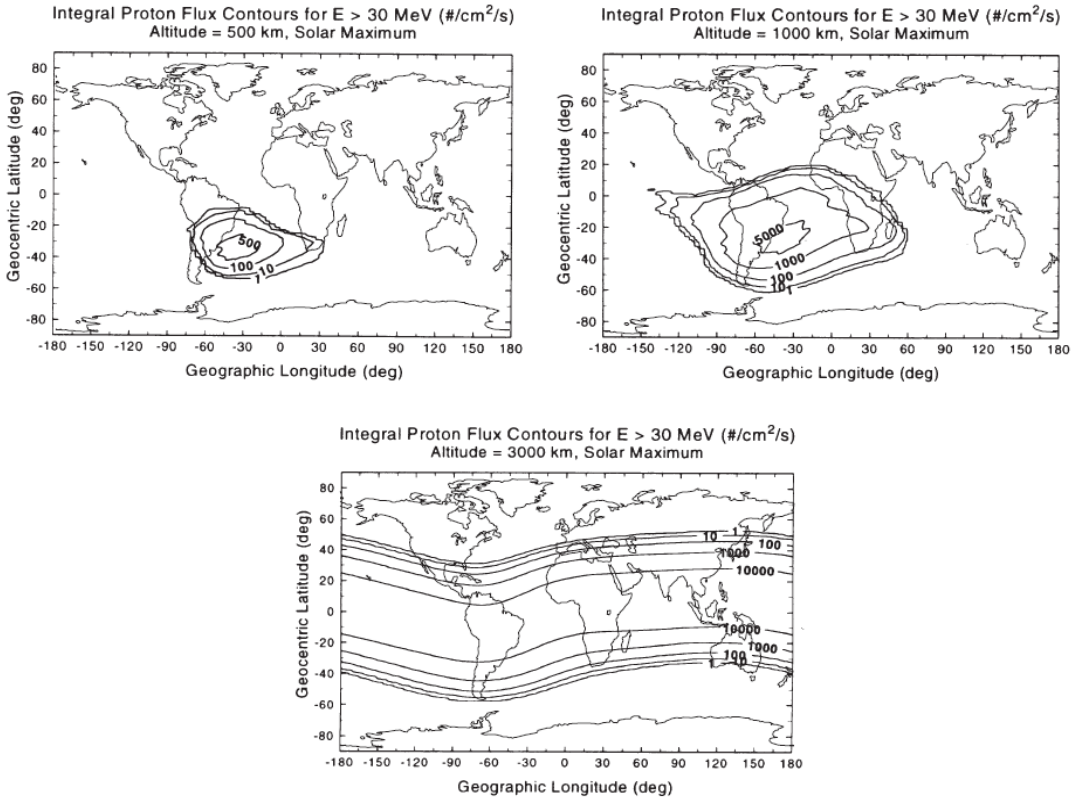


Figure 1.10 – Flux de protons en fonction de la latitude et longitude à 500 km, 1 000 km et 3 000 km d'altitude

Les protons peuvent interagir avec les atomes des matériaux du composant lorsque leur énergie est supérieure à quelques dizaines de MeV. Ils sont cent fois moins nombreux que les neutrons et leur faible abondance permet de considérer leur effet comme négligeable dans la perturbation du fonctionnement des circuits intégrés par rapport à celui des neutrons.

Les neutrons n'étant pas chargés, ils sont très pénétrants dans l'atmosphère et perdent peu d'énergie au contact des molécules présentes dans l'atmosphère. Ils arrivent au niveau du sol et

peuvent interagir avec les atomes constitutifs des différentes couches des composants électroniques créant des particules ionisantes qui peuvent induire des défaillances.

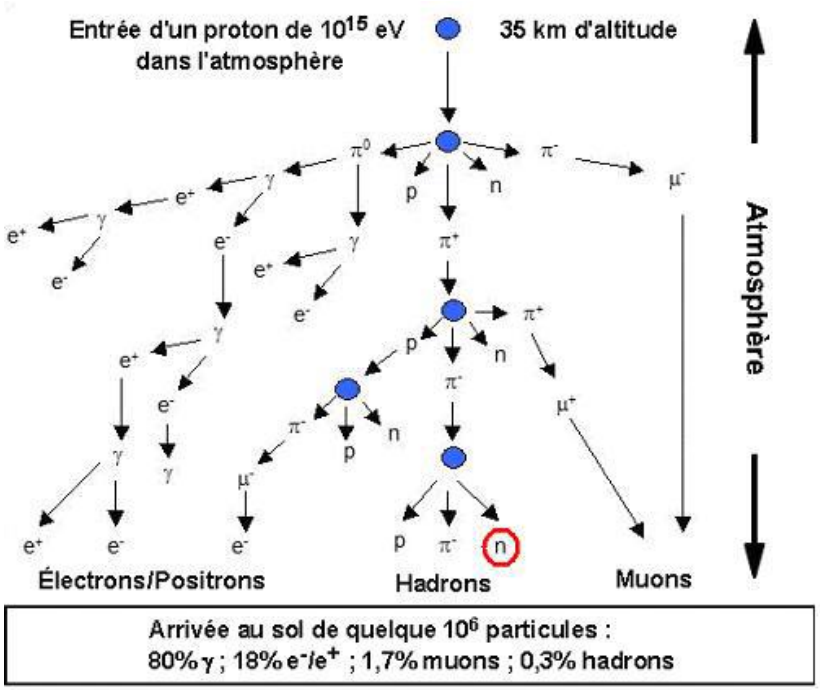


Figure 1.11 – Représentation schématique de la cascade de particules dans l'atmosphère terrestre

Jusqu'aux années 90, seuls les neutrons d'énergie supérieure à 100 MeV étaient une menace pour les composants de circuits intégrés. Mais pour les technologies actuelles, les neutrons d'énergie inférieure à 100 MeV posent eux aussi des problèmes, en raison de l'augmentation de la finesse de gravure des transistors. Les neutrons dits thermiques (neutrons de très faible énergie appelés aussi neutrons lents ont une énergie cinétique inférieure à 0,025 eV et une vitesse inférieure à 2190 m/s) sont pris aussi en considération dans la création d'événements singuliers (Single Event Effects - SEE). En effet, à basse énergie de nombreuses réactions de capture de neutrons deviennent beaucoup plus probables et se traduisent par la fission de certains matériaux en créant des particules secondaires chargées. Ces neutrons thermiques peuvent interagir avec le Bore qui est utilisé comme dopant ou constituant des couches de passivation [25][26].

Les radiations d'origine cosmique, au sol, sont issues de particules de 6ème ou 7ème génération. Les particules ont une direction hautement verticale, et sont soumises au cycle d'activité solaire. Les événements solaires peuvent augmenter localement le flux de particules de plus de 5000 % [27].

Le flux moyen de neutrons au sol est théoriquement estimé de l'ordre de 36 particules/cm^2/heure, mais dépend de plusieurs facteurs, dont l'altitude - à 3700 m le flux est multiplié par un facteur 13 - la latitude géomagnétique, le cycle solaire ainsi que le blindage environnant. Néanmoins, la référence prise en général est la ville de New York avec un flux d'environ 14

neutrons/cm^2/heure. La Figure 1.12 montre le spectre des neutrons atmosphériques en fonction de leur énergie mesuré à New York au niveau du sol, le flux différentiel s'étend de quelques eV à 1 GeV [28].

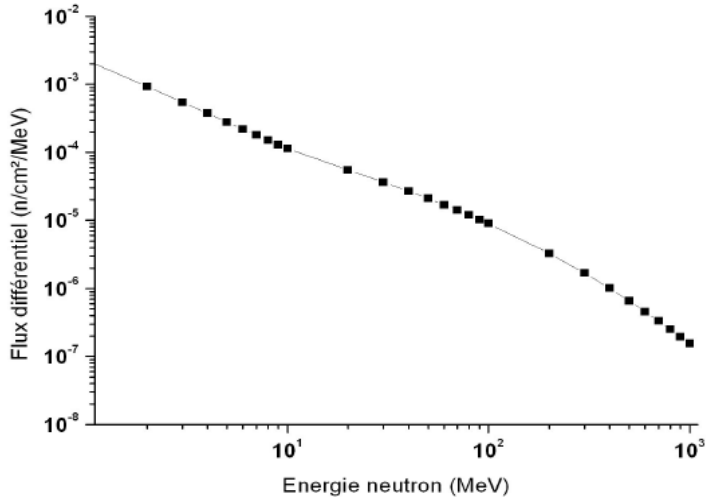


Figure 1.12 - Flux différentiel des neutrons atmosphériques induits par le rayonnement cosmique en fonction de l'énergie pour des conditions de référence selon [29]

1.3 Effets des Radiations sur les Circuits Intégrés

Les radiations peuvent causer des dommages permanents, transitoires dans les matériaux qu'elles traversent. Cette section donnera un aperçu des mécanismes d'interaction particule-matière. Après l'interaction, on peut classifier deux types d'effets distincts : la dose, qu'elle soit ionisante ou non, et les effets singuliers (causés par une unique particule).

1.3.1 Mécanismes d'interaction particule-matière

La quantité d'énergie perdue par une particule incidente lors de son passage dans la matière est en relation directe avec les défaillances permanentes ou transitoires. Cette quantité perdue, s'appelle *pouvoir d'arrêt total*, et est exprimée par la quantité d'énergie perdue par longueur de matériau traversé. Le *pouvoir d'arrêt total* est le résultat de deux phénomènes qui vont ralentir la progression de la particule incidente dans la matière. Ces phénomènes sont les pertes d'énergies électroniques et les pertes d'énergies nucléaires. Donc, le pouvoir d'arrêt total peut être décrit comme la somme de ces deux phénomènes :

$$\left(\frac{dE}{dx}\right)_{total} = \left(\frac{dE}{dx}\right)_{electronique} + \left(\frac{dE}{dx}\right)_{nucleaire} \quad (1.1)$$

Le premier phénomène, la perte d'énergie électronique, est dû aux interactions entre la particule incidente et les électrons des atomes du milieu traversé. L'interaction avec un électron peut conduire ce dernier à quitter son orbitale atomique, avec la possibilité d'aboutir à la création de paires électrons-trous le long du parcours de la particule. Dans la communauté spatiale, ce pouvoir d'arrêt électronique est appelé *transfert linéique d'énergie* (LET – Linear Energy Transfer).

Le deuxième phénomène, les pertes nucléaires, est dû aux collisions entre la particule et les noyaux des atomes du milieu traversé. Le résultat possible est l'éjection du noyau d'un réseau cristallin. Des défauts sont alors créés dans le réseau. La perte d'énergie associée à ce type de dommage est appelée perte d'énergie non ionisante (NIEL - Non Ionising Energy Loss).

La Figure 1.13 montre la contribution des deux phénomènes d'arrêt énoncés ci-dessus d'un ion Aluminium dans de l'aluminium, en fonction de l'énergie des ions.

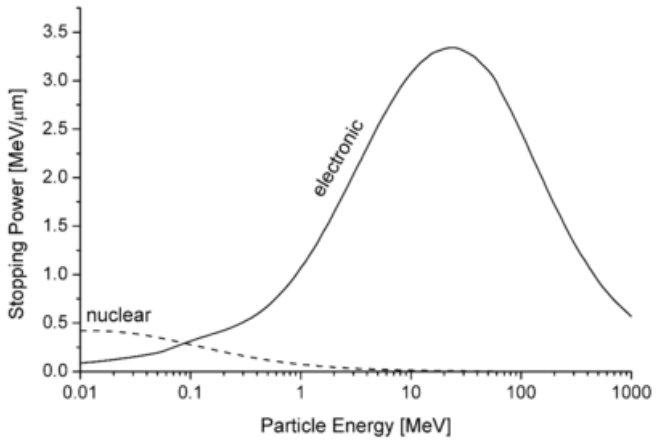


Figure 1.13 – Pouvoir d'arrêt électronique et pouvoir d'arrêt nucléaire d'un ion d'Aluminium

1.3.2 Les effets de dose

La dose est définie comme l'énergie déposée dans un matériau par unité de masse et s'exprime en gray (*Gy*). Un Gray correspond à l'absorption d'un Joule par kilogramme de matière. Généralement, une partie signifiante des effets des radiations que l'on peut observer est regroupée dans *les effets de dose*.

La dose ionisante est caractérisée quand une particule traverse un matériau et que l'énergie perdue par la particule est transférée aux électrons du matériau traversé. Par contre, si les atomes sont déplacés, on parle alors de dose non ionisante.

La dose ionisante, pour un échantillon de matière de surface *S*, d'épaisseur *dx*, de masse volumique ρ, irradié par une fluence de particules Φ déposant une énergie *dE*, est exprimée par [27] :

$$D_{ionisante} = N_{particules} \frac{dE}{masse}$$
$$D_{ionisante} = \Phi.S.\frac{dE}{S.dx.\rho} \quad (1.2)$$
$$D_{ionisante} = \Phi \frac{1}{\rho}\left(\left(\frac{dE}{dx}\right)_{eletrique} + \left(\frac{dE}{dx}\right)_{nucleaire}\right)$$

La dose non ionisante est définie comme le produit de la fluence de particules et la perte d'énergie non ionisante :

$$D_{nionisante} = \Phi.NIEL \quad (1.3)$$

Dans les sous-sections suivantes sont décrits les effets de doses ionisantes, résultantes de plusieurs mécanismes, et les effets de doses non ionisantes.

1.3.2.1 Dose Ionisante

Lors d'une irradiation, l'énergie perdue par les particules peut être transférée aux électrons du matériau traversé. Une partie de cette énergie entraîne la création de paires, issues de la libération de charges fixes fortement liées. La densité de paires électrons-trous, générée par seconde, par un rayonnement ionisant peut être considérée comme proportionnelle à l'énergie déposée. Cette relation de proportionnalité est exprimée par :

$$G = b\frac{dD}{dt} \quad (1.4)$$

où la constante *b* représente un coefficient multiplicateur qui exprime la densité de paires créées par *Gy* (gray) et par cm^{-3}, et *dD/dt* représente le taux de dose (exprimé en $Gy.s^{-1}$).

Selon la nature du matériau, les électrons peuvent atteindre la bande de conduction et libérer des trous dans la bande de valence [30]. L'énergie moyenne (E_p) cédée par le rayonnement ionisant, nécessaire à la création d'une paire, est supérieure à celle strictement nécessaire à sa création (énergie du gap, E_g). Dans les semi-conducteurs, l'énergie nécessaire à la création d'une paire électron-trou E_p est exprimée par la relation empirique [16] suivante :

$$E_p(eV) = 2{,}67E_g(eV) + 0{,}87 \quad (1.5)$$

Dans le cas du silicium, pour lequel E_g= 1,12 *eV*, il faut en moyenne 3,86 *eV* pour faire migrer un électron de la bande de valence à la bande de conduction. Dans l'oxyde de silicium (SiO_2), cette valeur devient 18 *eV*. Pour fixer un ordre de grandeur, 1 Gy dans 1 cm^3 de silicium génère 7,3 10^{14} paires électrons-trous et 1.5 10^{14} paires dans l'oxyde de silicium. Pour le cas du silicium non ou très

légèrement dopé, toute charge piégée qui n'est pas libérée thermiquement reviendra à l'équilibre par recombinaison. Donc, l'effet sera transitoire et disparaître avec l'irradiation. Il n'en est pas de même pour les isolants qui, à l'équilibre, n'ont pas la densité de porteurs libres qui permettraient le retour à l'équilibre par recombinaison des charges à cause de la très faible mobilité et reste donc piégée sur des niveaux parasites introduits par les impuretés ou les défauts dans le large gap.

Les mécanismes qui affectent les composants électriques sont présentés de façon non exhaustive par la suite :

- **Recombinaison**

  Après l'étape de génération de charges, que nous avons abordée précédemment, suit une phase de recombinaison initiale entre une partie des trous et des électrons créés par l'irradiation.

  Le processus de recombinaison initial dépend également de la valeur du champ électrique appliqué qui va plus ou moins séparer les porteurs entre eux. Pour un champ électrique faible, les porteurs ne seront pas séparés et vont donc pratiquement tous se recombiner. Au contraire, un fort champ empêchera toute recombinaison [31].

  La Figure 1.14 montre la mesure de la fraction de porteur non-recombinée dans l'oxyde de silicium en fonction du champ électrique appliqué pour des particules incidentes de natures et énergies différentes [32].

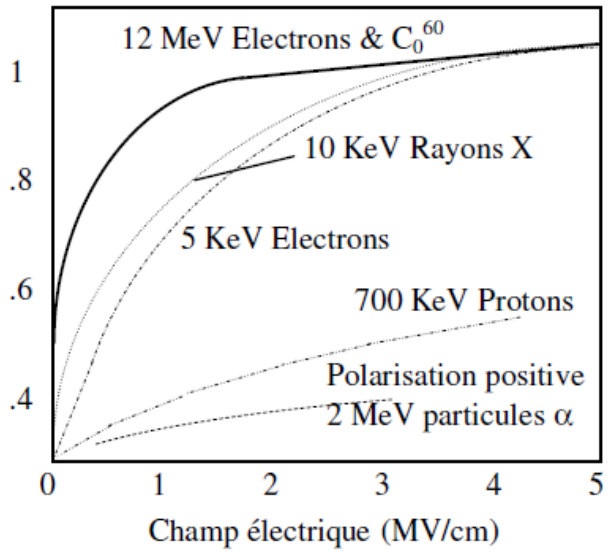


Figure 1.14 – Fraction non-recombinée dans l'oxyde de silicium (SiO_2) en fonction du champ électrique

- **Transport des trous**

  La mobilité des électrons est plus élevée que celle des trous, ainsi ils sont évacués plus rapidement de l'oxyde que les trous. Les trous ont tendance à rester sur place et le plus souvent à se piéger.

La vitesse à laquelle s'effectue le transport des trous, dépend de la température, du champ électrique et du processus de fabrication de l'oxyde [33][34][35].

- **Piégeage**

Les charges piégées dans l'oxyde peuvent être positives ou négatives et sont dues au piégeage de trous et d'électrons dans le volume de l'oxyde. Ce type de piégeage est la conséquence d'une irradiation du composant (excitation extérieure).
Les effets de dose ionisante citées précédemment sont résumés dans la Figure 1.15 [7].

Pour les circuits CMOS, les caractéristiques électriques sont liées à l'accumulation de charges dans les oxydes ainsi qu'à l'interface oxyde/semi-conducteur et la création d'un canal permanent. Ceci pourra engendrer plusieurs conséquences dont les plus significatives sont la dérive des tensions de seuil (définie comme la valeur de la tension à appliquer, entre la grille et la source, pour obtenir un courant de drain non nul) et l'augmentation du courant de fuite.

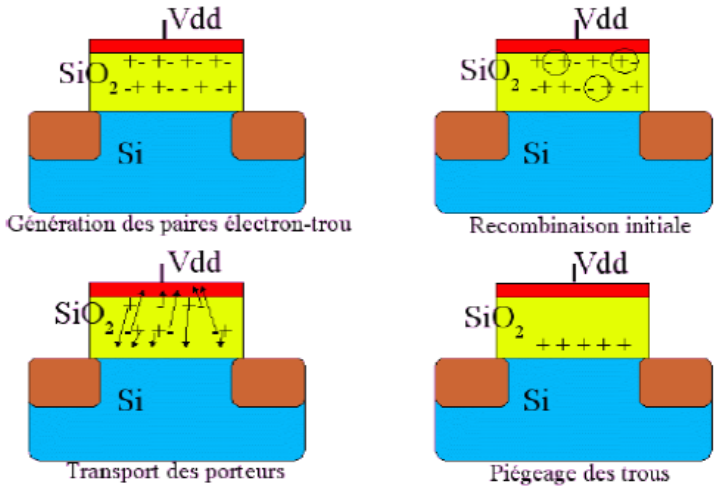


Figure 1.15 – Les effets de dose ionisante [7]

1.3.2.2 Dose Non Ionisante

L'introduction de dose non ionisante dans le réseau cristallin du matériau touché va créer des nouveaux pièges modifiant ainsi les caractéristiques de fonctionnement du circuit. Parmi les effets de dose non ionisante on peut citer cinq types de contributions :

- ✓ Augmentation du courant de fuite
- ✓ Réduction de la durée de vie des porteurs minoritaires
- ✓ Piégeage de porteurs
- ✓ Compensation des dopants
- ✓ Effet tunnel

1.3.3 Les événements Singuliers

Les événements singuliers (Single Event Effects - SEE) on été observés en premier lieu au sein des composants eux-mêmes (pollution des boîtiers céramiques par du Thorium radioactif, années 70, puis dans le domaine des applications spatiales, années 80). A cette époque les composants électroniques n'étaient sensibles qu'aux effets des particules lourdes et/ou très énergétiques (ions lourds). La miniaturisation des composants électroniques les rend de plus en plus sensibles aux particules de faible masse tels les protons et les neutrons, ainsi qu'aux particules ayant basses énergies.

Les évènements singuliers ont la même origine phénoménologique que la dose ionisante, c'est à dire la création de paires de porteurs. Cependant les conséquences et le traitement sont différents. Ces conséquences peuvent s'étendre à l'ensemble d'un système sous la forme d'une perturbation ou d'une panne fonctionnelle réversible ou permanente. Les événements singuliers sont caractérisés par une injection de charge élevée et localisée le long de la trajectoire de la particule incidente. Les SEE sont le résultat, principalement, de la déposition et de la collection de charges sur un nœud sensible du circuit. Ce nœud est une jonction en polarisation inverse qui est à l'intérieur de la zone de déplétion, le champ électrique permet alors une collection des charges excédentaires [27].

Les ions qui traversent ce volume produisent une colonne d'électrons-trous qui peut engendrer un événement singulier. C'est donc un mécanisme direct. Pour ce qui est des protons et des neutrons, ils transfèrent une partie ou la totalité de leur énergie lors des collisions. Les produits de ces collisions peuvent ioniser la matière et provoquer un événement singulier. C'est un mécanisme indirect.

Deux phénomènes participant à la génération d'un événement singulier: la génération de charges, puis la collection de ces charges [36].

1.3.3.1 Génération de Charges

Pour quantifier les charges déposées par le passage d'un ion, on peut utiliser de la même façon que pour la dose ionisante, le transfert linéique d'énergie (LET). Le LET d'une particule en fonction de la distance parcourue dans la matière est donné par la courbe de Bragg. Dans La Figure 1.16 on peut voir la courbe de Bragg pour des particules alpha dans l'air. Au cours de la majeure partie du parcours de la particule, le LET reste relativement constant. Vers la fin du trajet, lorsque la particule a perdu presque toute son énergie, le LET augmente subitement pour ensuite s'annuler brutalement lorsque la particule incidente est au repos. Ce phénomène est appelé le pic de Bragg.

L'énergie déposée par la particule dans un volume de densité ρ et de longueur de trajet d peut être calculée par :

$$E_{dep} \approx LET \times d \times \rho \tag{1.6}$$

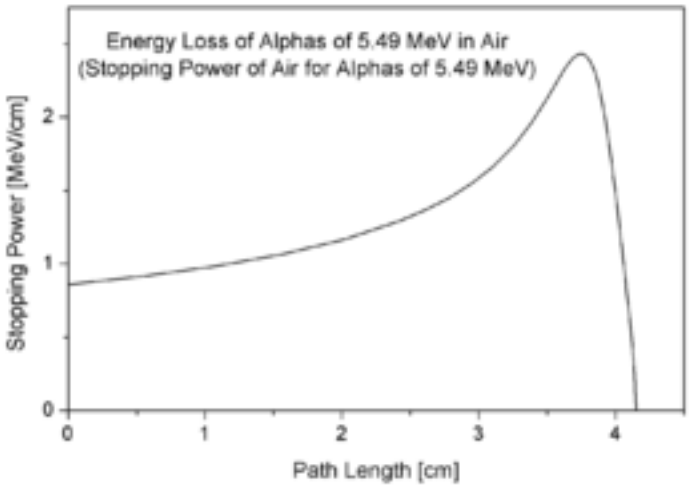


Figure 1.16 - Courbe de Bragg pour les particules Apha dans l'air

1.3.3.2 Collection de charges

Les SEE sont caractérisés par une injection de charge élevée et localisée le long de la trajectoire de la particule incidente. Lors du passage de la particule, deux phénomènes sur des échelles de temps différents participent à collection de charges:

- Conduction (*drift*): les électrons, sous l'influence du champ électrique, sont déplacés vers le drain du transistor. Le courant est responsable de la réponse initiale du nœud touché. La durée de ce phénomène est inférieure à la nanoseconde.

- Diffusion: Les électrons générés dans le substrat peuvent diffuser vers le drain et ainsi être collectés. Ce phénomène peut durer près de plusieurs nanosecondes.

Un troisième phénomène peut apparaître lorsque l'ion sort de la zone de déplétion. Cet ion distord le champ électrique et l'étire en dehors de la zone de déplétion. La longueur de funneling n'est pas toujours bien définie. La Figure 1.17 illustre les trois phénomènes.

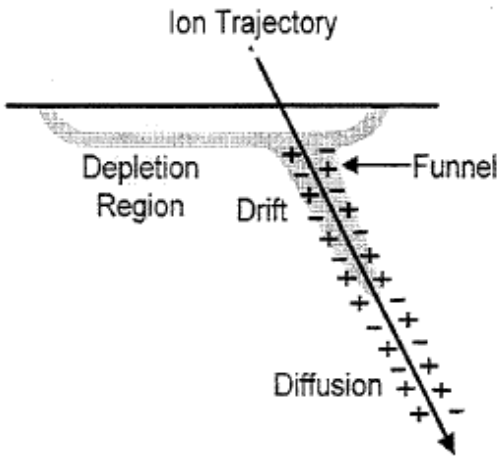


Figure 1.17 - Phénomènes de collection de charges: conduction (*drift*), *funneling* et diffusion

Comme conséquence de l'impact d'une particule ionisante, différents événements peuvent se produire. Ces événements sont les suivants (cette liste n'est pas exhaustive) :

1.3.3.3 SET Single-Event Transient

Le SET est le premier effet singulier direct et est lié à la collection de charge dans un noeud sensible. La particule incidente induit par ionisation une quantité de charge. Cette charge est collectée par le champ électrique et résulte en la création d'un photocourant. Les conséquences de cet événement sont la génération de signaux transitoires indésirables qui peuvent perturber le fonctionnement des systèmes numériques et analogiques.

1.3.3.4 SEU Single Event Upset

Le Single Event Upset est le basculement du contenu d'un bit dans une bascule ou cellule mémoire. Cette erreur induite par radiation est causée lorsqu'une particule chargée dans un circuit perd de l'énergie en ionisant le milieu dans lequel il traverse, laissant lors de sa trajectoire un sillage de paires électrons-trous [37]. La particule produit des paires électrons-trous le long de sa trajectoire, ce qui fait apparaître une impulsion de courant laquelle, si l'amplitude et la durée sont suffisantes, peut produire le même effet que celui d'un signal extérieur appliqué au transistor. Le Single Event Upset, appelé aussi *soft-error* ou *bit-flip*, est un événement non destructif qui peut générer des erreurs dans le contenu des cellules mémoires (SRAMs, registres, flip-flops, etc), mais l'élément contenant le bit corrompu reste opérationnelle pour les opérations futures de lecture-écritre.

Les conséquences des SEUs sur le comportement des circuits intégrés dépendent de la nature de l'information contenue dans les cellules perturbées et de l'instant d'occurrence. En fonction des cellules qui ont été perturbées et de l'instant d'occurrence, dans certains cas les SEU peuvent avoir des

conséquences critiques, comme l'arrêt du système suite à la perte de séquencement de son processeur [38].

1.3.3.5 Multiple Cell Upsets (MCU) et MBU Multiple Bit Upsets (MBU)

Ces événements se produisant quand une particule affecte plusieurs structures voisines de type "bascule" en y induisant des changements d'état. Dans le cas du MBU, les cellules mémoire perturbées se trouvent dans un même mot mémoire et la conséquence peut être critique car la faute peut échapper aux techniques de détection/correction d'erreurs implémentées (codes de Hamming par exemple). La probabilité de ce type d'événement singulier est plus faible que celle du SEU mais elle augmente avec la miniaturisation de l'électronique.

Cet événement a comme conséquence la complexification de la problématique de la détection et de la correction d'erreurs dans les systèmes numériques.

1.3.3.6 SEL Single Event Latch-up

Le Latch-up est une condition de courant élevé qui résulte de la mise en conduction d'un thyristor parasite (SCR, Silicon-Controlled Rectifier) présent dans des structures à 4 couches (par exemple dans les circuits intégrés CMOS). Un court-circuit entre l'alimentation et la masse du circuit est créé, et peut aboutir à la destruction du composant si l'alimentation n'est pas coupée [39][27].

1.3.3.7 SHE Single Hard Error

Ce phénomène a pour conséquence un bit collé. La particule provoque un basculement irréversible d'un point mémoire.

1.3.3.8 SEFI Single Event Functional Interrupt

Le SEFI se traduit par le blocage de la fonctionnalité du composant en le faisant passer dans des états interdits. Il concerne notamment les machines d'état des composants numériques. La récupération de l'état normal du circuit passe souvent par un reset complet, voir par un redémarrage de l'alimentation.

1.3.3.9 SEGR Single Event Gate Rupture

Il s'agit de la rupture de grille d'un composant MOS après qu'il ait été irradié. Les SEGR sont déclenchés par des ions lourds lors de leur passage au travers de la couche isolante [40]. Ces défauts ne sont pas électriquement actifs immédiatement après l'irradiation. Ils peuvent être activés après la fin de l'irradiation et sont alors identifiés comme des défauts précurseurs conduisant au déclenchement du claquage prématuré de l'oxyde de grille. Ils sont toujours destructifs pour les dispositifs.

1.3.3.10 SEB Single Event Burnout

Le SEB est une condition qui peut provoquer la destruction du dispositif dû à un courant élevé dans un composant de puissance. Le SEB provoque l'échec définitif du dispositif. Ce phénomène s'observe principalement dans les composants de puissance qui contiennent des centaines de transistors en parallèle. Les dommages d'un seul transistor peuvent conduire à la destruction de ses voisins par *effet d'avalanche* et rendre le circuit inutilisable.

1.4 Conclusion

Dans ce chapitre a été décrit l'environnement radiatif et la problématique liée aux effets des radiations sur les circuits intégrés ainsi qu'un aperçu des différents effets engendrés par une particule incidente impactant une partie sensible de circuit.

Le progrès permanent de la technologie de fabrication des circuits intégrés conduit à des circuits potentiellement plus sensibles aux effets du rayonnement naturel présent dans l'environnement où ils fonctionnent [1][2]. Comme décrit dans ce chapitre, l'impact de particules énergétiques (neutrons, particules alpha, ions lourds, ...) avec une zone sensible d'un circuit intégré peut avoir une large palette de conséquences rassemblées sous le sigle SEE (Single Event Effects). Parmi les SEE cette thèse s'intéresse aux effets de type SEU ayant lieu dans des circuits intégrés tels que les processeurs et les composants de type SRAM. Lorsque l'objet de l'étude est un circuit de type processeur qui est capable d'exécuter un jeu d'instructions ou de commandes, les conséquences d'un SEU peuvent être critiques et dans les cas extrêmes peuvent mener à la perte de contrôle du système. Le très grand nombre de cellules mémoire inclues dans les circuits programmables tels que les processeurs, FPGA, systèmes sur puce, réseaux sur puce, rend obligatoire envisager le SEU comme une source d'erreurs potentielles dans l'application finale, pour toute application pour laquelle de telles erreurs transitoires peuvent avoir des conséquences critiques.

Cette thèse a pour but l'étude de la fiabilité d'un algorithme auto-convergeant face aux SEU's, c'est à dire, explorer l'efficacité et la robustesse intrinsèque de cet algorithme qui peut être nécessaire dans l'avenir des processeurs *multicore* pour la communication entre ses *cores*. Ces processeurs peuvent partager la charge de travail par la distribution des tâches sur plusieurs de leurs *cores* en vue d'accomplir cela, cependant ces *cores* ont besoin de communiquer les uns avec les autres de façon fiable. La distribution des tâches sur les *cores* peut être vue dans le contexte de systèmes distribués. Un système distribué mal initialisé ou perturbé retournera automatiquement à un fonctionnement correct en un nombre fini d'étapes de calcul. Ce retour automatique est du à la propriété de l'autostabilisation qui est un concept de tolérance aux fautes dans ces systèmes. Dans la suite de ce

manuscrit sont donnés des aperçus des systèmes distribués et de l'auto-stabilisation. L'algorithme d'auto-convergence utilisé au cours de cette thèse est présenté ainsi que ses problèmes potentiels dus aux fautes de type SEU.

Chapitre 2. Les systèmes distribués, l'autostabilisation et l'algorithme d'auto-convergence

Ce chapitre propose un aperçu sur les systèmes distribués ainsi que sur l'autostabilisation et l'algorithme d'auto-convergence étudié au cours de cette thèse. Ce chapitre s'articule autour de sept sections. Nous commencerons par discuter les motivations : pourquoi utiliser des systèmes distribués et la contribution et l'objectif de ces travaux. Les généralités sur les systèmes distribués sont présentées : une comparaison entre les systèmes distribués et les systèmes centralisés ainsi que les caractéristiques propres des systèmes distribués. Puis nous présenterons la tolérance aux fautes, c'est à dire les critères pour la classification des fautes dans les systèmes distribués et l'approche pour la tolérance aux fautes ainsi que les défis de la mise en œuvre des systèmes distribués et leur modélisation. Une section est consacrée à l'approche d'autostabilisation, les propriétés qui doivent être satisfaites pour que le système soit autostabilisant et l'autostabilisation dans la pratique. Dans la dernière section, nous présenterons l'algorithme d'autoconvergence ainsi que ses problèmes potentiels face aux SEU.

2.1 Motivations pour les systèmes distribués

Les constants progrès technologiques des ordinateurs et le haut débit permettent aux différentes applications sur des ordinateurs distincts et distants de coopérer pour effectuer des tâches coordonnées. Avant les années 80, les ordinateurs étaient encombrants et chers mais à partir de la mi-80, l'avancement de la technologie a changé cette situation permettant le développement de microprocesseurs moins chers et très puissants et des réseaux informatiques de haute vitesse tels que:

- LANs (Local Area Networks) qui permet à des centaines d'ordinateurs à l'intérieur d'un bâtiment d'être connectés, ce qui permet l'échange d'informations.
- WAN (Wide Area Networks) qui permet à des millions d'ordinateurs dans le monde d'être connectés.

L'une des raisons principales de l'utilisation d'un système distribué est le partage de ressources (logicielles, matérielles, données, services, etc.) puisque c'est économiquement intéressant. Une autre raison est le rapport coût/performance, car c'est mieux utiliser plusieurs processeurs interconnectés qu'un seul ordinateur centralisé.

Un système distribué avec la propriété d'autostabilisation : suite à tout incident qui change l'état du système, l'autostabilisation assure de retrouver automatiquement, après un certain temps de fonctionnement sans nouvel incident, une exécution légitime, et donc un fonctionnement correct. En particulier, ceci permet de tolérer les fautes transitoires : modification arbitraire de l'état d'un processus au cours d'une exécution. Une telle faute peut apparaître comme conséquence d'un évènement singulier, SEE, provoqué par l'interaction d'une unique particule chargée avec une zone sensible du circuit intégré considéré. Cette thèse s'intéresse particulièrement aux évènements de type SEU (Single Event Upset) qui résultent en la modification du contenu d'un bit d'un élément mémoire. Une telle faute transitoire peut laisser le système dans une configuration quelconque, totalement imprévisible et le retour automatique à un état légitime est particulièrement souhaitable dans un système, un satellite par exemple, sur lequel une intervention extérieure est difficile ou même pas possible.

2.2 Les généralités sur les systèmes distribués

Un système distribué en informatique, est une collection d'unités de calcul indépendantes qui sont connectées entre-elles et qui apparaissent à l'utilisateur comme un seul système cohérant. Ces unités de calcul, aussi appelées processeurs, coopèrent via des échanges de messages dans le but de réaliser une tâche commune. Ces messages transitent via des liens ou canaux de communications bidirectionnels, mais dans certains cas, les liens de communications peuvent être supposés unidirectionnels. Dans ce qui suit sont données les caractéristiques propres des systèmes distribués et est faite une comparaison entre les systèmes distribués et les systèmes centralisés.

2.2.1 Comparaison entre les systèmes distribués et les systèmes centralisés

Un système centralisé est constitué de plusieurs entités communicantes, ne pouvant fonctionner qu'en faisant référence à un site particulier, investi d'un rôle plus important que les autres. Un système centralisé peut être comparé à un chef d'orchestre qui assure le fonctionnement de l'ensemble du système.

Les systèmes distribués sont nés de la connexion des ordinateurs entre eux. Dans ces systèmes la distribution des données et des unités de calcul est subie, c'est à dire, certains programmes de calcul sont parallélisés parce qu'ils nécessitent plus de mémoire que celle qui peut offrir une seule machine. Contrairement, certaines applications sont distribuées dans le seul but d'augmenter leur robustesse.

Mais de façon générale, la distribution des utilisateurs et de données dans une application distribuée n'est pas un choix du programmeur contrairement à la programmation parallèle : c'est un état de fait qu'il doit prendre en compte. Le but du domaine des systèmes distribués est de masquer la distribution en apparaissant comme un seul système cohérant, et ainsi faciliter l'utilisation du système. Mais afin de pouvoir utiliser plusieurs machines pour résoudre un problème donné, il faut encore avoir des programmes qui le permettent. L'algorithmique distribuée est l'étude de ces problèmes. Un algorithme distribué est constitué de l'ensemble des algorithmes des processus du système. Le programme de chaque processus est constitué d'actions qui sont soit la réception de messages, soit l'émission d'une information vers un autre processus, soit l'exécution d'une séquence d'instructions internes. Mais il n'est pas si simple de concevoir des algorithmes distribués performants, c'est-à-dire qu'ils permettent d'obtenir le résultat voulu en un temps fini, effectuant un minimum d'échanges d'information et utilisant peu d'espace mémoire. Enfin, les systèmes distribués diffèrent des systèmes centralisés en trois aspects [41] :

- Le manque de connaissances sur l'état global (localité des informations) : dans un système centralisé, un processus peut connaître les valeurs de toutes les variables (ou états) utilisées par tous les autres processus. Dans un système distribué, un processus n'a la possibilité de connaître que les états qui lui sont envoyés par les processus auxquels il est directement connecté (aussi appelé voisins). Cependant, ces informations peuvent avoir été modifiées pendant le temps de la transmission et donc être obsolètes lorsque le processus destinataire en prendra connaissance.
- L'absence de temps global (localité de temps) : dans un système centralisé, les exécutions se font séquentiellement. Dans un système distribué, chaque processus exécute des actions selon une relation d'ordre partiel et non total. D'une manière générale, entre deux processus exécutés en parallèle, nous ne savons donc pas lequel se terminera en premier.
- Non déterminisme : tous les composants mis en jeu dans un système distribué ne fonctionnent pas forcément à la même vitesse. Il est donc impossible de déterminer à l'avance le comportement global d'une suite de processus d'un algorithme distribué.

Les contraintes principales des systèmes distribués sont: le manque de connaissance globale sur le système et l'absence de temps global [42]. Cependant, la problématique majeure des systèmes distribués est donc de décrire formellement les mécanismes permettant de résoudre une tâche dont les données sont réparties dans le réseau tout en tenant compte de ces contraintes. L'algorithme distribué décrit les traitements - envois de messages, réceptions de messages et calculs internes - que les processeurs doivent exécuter pour résoudre une tâche distribuée. La spécification est un énoncé formel de la tâche à résoudre. Un algorithme distribué a une tâche à résoudre et cette tâche peut avoir des spécifications, dans ce cas, les spécifications sont généralement formulées en deux propriétés : la sûreté et la vivacité. La sûreté d'une spécification correspond à l'ensemble des propriétés qui doivent

être vérifiées en permanence durant une exécution d'un protocole. La vivacité d'une spécification correspond à l'ensemble des propriétés qui doivent être vérifiées à certains instants de l'exécution.

2.2.2 Caractéristiques propres des systèmes distribués

Comme dit précédemment un système distribué se distingue par un certain nombre de caractéristiques qui montrent ses avantages par rapport au système centralisé. Un système centralisé est constitué d'une seule machine, i.e., dans un système centralisé tout est localisé sur la même machine et accessible par le programme. Un système distribué est constitué de plusieurs machines, ce système a comme caractéristiques principales : le partage des ressources, la distribution des données, l'échange d'informations, la tolérance aux fautes et la croissance de la puissance de calcul. Les sections suivantes décrivent chacune de ces caractéristiques.

2.2.2.1 Le partage des ressources

Un réseau est un système avec plusieurs utilisateurs et tâches. Le partage des ressources permet au système l'optimisation de ces ressources entre plusieurs utilisateurs/machines avec l'accès à distance à une ressource indisponible en local et/ou l'accès aux mêmes ressources depuis tous les endroits du système. Cette caractéristique est économiquement intéressante pour de tels systèmes tout en essayant de conserver une qualité de service semblable à celle d'un système avec un seul utilisateur. Cependant, cette caractéristique pose le problème de l'accès concurrent des utilisateurs ou des applications à des ressources.

2.2.2.2 La distribution des données

Plusieurs machines connectées constituent un réseau qui permet d'obtenir un espace disque conséquent pour un coût raisonnable. L'espace en disque d'une seule machine est généralement insuffisant pour stocker l'ensemble des applications et des données disponibles dont un utilisateur a besoin, avoir plusieurs machines en réseau permet d'avoir plus d'applications, de données et échanges de ces données que en fonction de la taille de réseau peut être à grande échelle. Comme un exemple nous pouvons mentionner le réseau *internet* qui permet la communication entre milliards d'utilisateurs et l'échange d'informations les plus variables.

2.2.2.3 La croissance de la puissance de calcul

Un système centralisé est constitué d'une seule machine avec un processeur qui prend un temps d'exécution important lors d'un calcul. Ce même calcul sur un système distribué aura son temps d'exécution réduit de manière significative puisque ces systèmes permettent les calculs concurrents. Dans un système distribué, les ordinateurs sont interconnectés entre eux et du point de vue de l'utilisateur le système apparaît comme un unique ordinateur virtuel très puissant. Ces systèmes

peuvent avoir différentes représentations de données, de code machine, d'interfaces pour les protocoles, langages de programmation, système d'exploitation, etc., i.e., ils sont hétérogènes. Dans le cas d'un calcul qui nécessite d'un temps d'exécution plus long il est possible l'utilisation d'un « intergiciel » (aussi appelé *middleware*, c'est la couche logicielle situé entre les couches basses, c'est à dire les systèmes d'exploitation, protocoles de communication, et les applications dans un système informatique réparti) qui s'occupe d'activer les composantes et de coordonner leurs activités de telle sorte qu'un utilisateur perçoive le système comme un unique système intégré.

2.3 Tolérance aux fautes

L'un des enjeux fondamentaux de l'algorithmique distribuée est la tolérance aux fautes. La tolérance aux fautes est la capacité d'un système à gérer et résister à un ensemble de problèmes. Une faute correspond à une défaillance temporaire ou définitive d'un ou plusieurs composants (processeurs ou canaux) du système. Un composant est dit défaillant lorsqu'il ne répond plus à ses fonctions. Le but des algorithmes tolérants aux fautes est de garantir le bon fonctionnement du système malgré les dysfonctionnements des différents composants du système.

2.3.1 Critères pour la classification des fautes dans les systèmes distribués

Les fautes dans les systèmes distribués sont généralement classés suivant trois critères [43]: le temps, la nature et l'extension. Dans ce qui suit est décrit chacun de ces critères.

- Temps : classifie les fautes ayant lieu dans des systèmes distribués en ce qui concerne la localisation dans le temps. Généralement, trois types de fautes possibles sont distingués:
    - *fautes transitoires* : fautes qui sont de nature arbitraire et peuvent perturber le système, mais il y a un point dans l'exécution au-delà duquel ces fautes ne se produisent plus.
    - *fautes permanentes* : fautes qui sont de nature arbitraire et peuvent perturber le système, mais il y a un point dans l'exécution au-delà duquel ces fautes se produisent toujours.
    - *fautes intermittenets* : fautes qui sont de nature arbitraire et peuvent perturber le système à tout moment durant l'exécution.
- Nature : Un élément du système distribué peut être représenté par un automate, dont les états représentent les valeurs possibles de variables de l'élément, et dont les transitions représentent le code exécuté par l'élément. Alors les fautes peuvent être distinguées selon qu'elles concernent l'état ou le code d'élément:

- *fautes liées au l'état* : des changements dans les variables de l'élément peuvent être causés par des perturbations de l'environnement (ondes électromagnétiques, par exemple), attaques (débordement de buffer, par exemple) ou tout simplement des fautes sur la partie de l'équipement utilisé. Par exemple, il est possible pour certaines variables avoir des valeurs qu'elles ne sont pas censées d'avoir si le système fonctionne normalement.
- *fautes liées au code* : un changement arbitraire dans le code d'un élément est le plus souvent résultat d'une attaque intentionnelle (le remplacement, par exemple, d'un élément par un opposant malveillant), mais certains types, moins graves, correspondent à des bugs ou à une difficulté à supporter la charge.

- Extension (espace) : l'extension des fautes, c'est à dire, le nombre des composants du système peut être frappé par des fautes ou des attaques.

Les fautes transitoires sont la base des algorithmes autostabilisants, c'est à dire une faute peut entraîner des comportements/conséquences arbitraires durant une certaine période, mais une fois que cette période (temps fini) est passée, l'algorithme reprend son fonctionnement normal. De ce fait, il n'est pas nécessaire d'examiner les causes et le déroulement de l'erreur, c'est à dire il suffit de garantir que même en partant d'un état erroné l'algorithme finira par donner le bon résultat.

Contrairement à un système centralisé, dans un système distribué, suite à une faute transitoire ou définitive d'un composant du système, une partie des services peut continuer à fonctionner. Dans un tel système, la distribution doit être transparente, c'est à dire le système doit être vu comme un tout et non comme une collection de composants (processeurs ou canaux distribués). La norme *ODP* (*Open Distributed Processing*) définit la *transparence de fautes* d'un système distribué comme la caractéristique qui permet au système fonctionner en présence de fautes, matérielles ou logicielles, sans que les utilisateurs sachent comment les fautes ont été surmontées. Comme un exemple nous pouvons citer un système d'émails qui peut retransmettre un message jusqu'à ce qu'il soit livré avec succès.

2.3.2 Approches pour la tolérance aux fautes: Les algorithmes robustes et autostabilisants

Lorsque des fautes se produisent sur un ou plusieurs des éléments qui composent un système distribué, il est essentiel de pouvoir les traiter. Si un système ne tolère aucune faute que ce soit, la défaillance d'un seul de ses éléments peut compromettre l'exécution de l'ensemble du système : c'est le cas pour un système dans lequel une entité a un rôle central (comme le DNS, Domain Name System). Le DNS est un service permettant de traduire un nom de domaine en informations de plusieurs types qui y sont associées, notamment en adresses IP de la machine portant ce nom. Afin de préserver la

durée de vie utile du système, plusieurs méthodes ad hoc ont été développées, qui sont généralement spécifiques à un type particulier de faute qui est susceptible de se produire dans le système en question. Cependant, ces solutions peuvent être classées selon que l'effet soit visible ou non par un observateur (un utilisateur par exemple). Une solution de *masquage* cache l'apparition de fautes à l'observateur, tandis que la solution *non masquage* ne présente pas cette caractéristique: l'effet des fautes est visible sur une certaine période de temps, puis le système reprend à se comporter correctement.

Une approche de masquage peut sembler préférable car elle s'applique à un plus grand nombre d'applications. L'utilisation d'une approche non-masquage pour réguler le trafic aérien rendrait les collisions possibles après l'apparition de fautes. Cependant, une solution de masquage est généralement plus coûteuse (en ressources et en temps) qu'une solution non-blocage, et ne peut tolérer les fautes à condition qu'elles ont été prévues. Pour les problèmes tels que le routage, où l'impossibilité de transporter des informations pendant quelques instants n'aura pas de conséquences catastrophiques, une approche non-masquage est parfaitement adaptée. Deux grandes catégories pour les algorithmes de tolérance aux fautes peuvent être distinguées [43] :

- Algorithmes robustes : ils utilisent la redondance sur plusieurs niveaux d'information, de communication, ou des nœuds du système, afin de se recouvrir dans la mesure où le reste du code sans risque est exécuté. Ils s'appuient généralement sur l'hypothèse selon laquelle un nombre limité de fautes va frapper le système, de manière à préserver au moins la majorité d'éléments corrects (parfois plus si les fautes sont plus sévères). Typiquement, ceux-ci sont des algorithmes de masquage.
- Algorithmes d'autostabilisation : ils reposent sur l'hypothèse que les fautes sont transitoires (en d'autres termes, limitées dans le temps), mais ne définissent pas de contraintes concernant la longueur (durée) des fautes (ce qui peut impliquer tous les éléments du système). Un algorithme est autostabilisant [44] [45] si il parvient, en un temps fini, à présenter un comportement approprié, indépendamment de l'état initial de ses éléments, ce qui signifie que les variables des éléments peuvent exister dans un état qui est arbitraire (et impossible à atteindre par l'exécution normale de l'application). En général, les algorithmes d'autostabilisation sont de type non-masquage, parce qu'entre le moment où les fautes cessent et le moment où le système est stabilisé pour un comportement approprié, l'exécution est susceptible d'être assez irrégulière.

Les algorithmes robustes sont assez proches de ce qu'est conçu intuitivement comme tolérance aux fautes. Par exemple, si un élément est sensible aux fautes, donc chaque élément est remplacé par trois éléments identiques, et chaque fois qu'une action est réalisée, l'action est effectuée trois fois par chacun des éléments, et l'action est en fait effectuée sur ce qui correspond à la majorité

des trois actions individuelles. L'autostabilisation semble être davantage liée à la notion de convergence en mathématiques ou la théorie de contrôle, où l'objectif est d'atteindre un point fixe quelle que soit la position initiale; le point fixe correspond ici à une exécution appropriée. Être capable de démarrer avec un état arbitraire peut sembler étrange (car il semblerait que les états initiaux des éléments sont toujours bien connus), mas des études [46] ont montré que si un système distribué est soumis à l'arrêt et les défaillances de nœuds de type redémarrage (qui correspond à une défaillance définitive suivie d'une réinitialisation), et les communications ne peuvent pas être totalement fiables, donc un état arbitraire du système peut effectivement être atteint. Même si la probabilité de l'exécution qui mène à cet état arbitraire est négligeable dans les conditions normales, il n'est pas impossible reproduire une telle exécution par une attaque sur le système . Dans tous les cas, quelle que soit la nature de ce qui conduit le système à cet état arbitraire, un algorithme auto-stabilisant est capable de fournir un comportement approprié dans une période finie de temps. En fait, les algorithmes distribués auto-stabilisants se trouvent dans un certain nombre de protocoles utilisés dans les réseaux informatiques [47][48].

La Figure 2.1 résume les capacités relatives des algorithmes auto-stabilisants et robustes, respectivement. Les trois axes prennent en compte les trois possibilités de classer les fautes dans les systèmes distribués qui ont été décrits précédemment. Avec un algorithme autostabilisant, un utilisateur externe peut rencontrer un comportement erratique (phase de stabilisation) après que les fautes aient effectivement cessé, tandis qu'un algorithme robuste apparaîtra toujours avec un comportement correct. En revanche, un algorithme autostabilisant ne fait aucune hypothèse sur l'extension ou la nature des fautes, tandis que les systèmes robustes généralement mettent des contraintes sur ceux-ci.

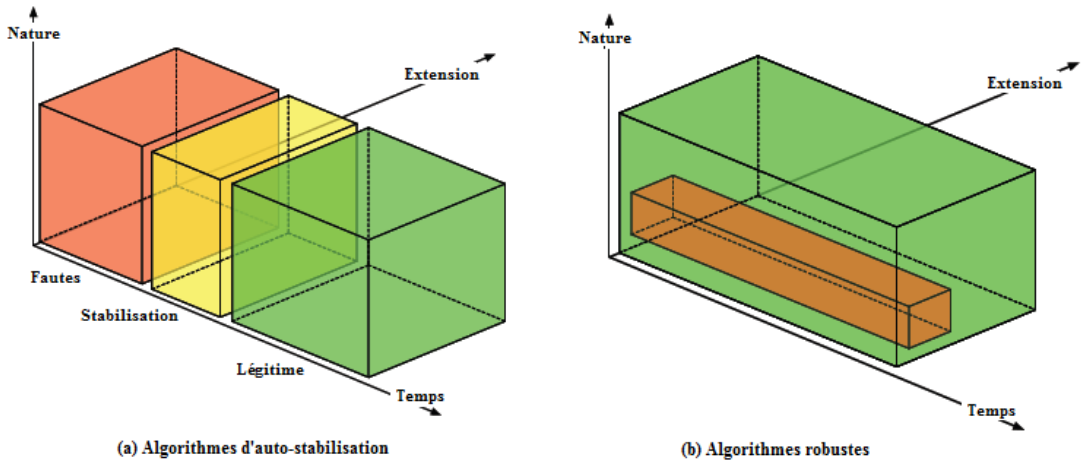


Figure 2.1 - Auto-stabilisation vs. Robustesse

2.4 Les défis de la mise en œuvre de systèmes distribués

Comme dit précédement, un système distribué peut être défini comme un réseau d'entités indépendantes de calcul ayant le même but commun : la réalisation d'une tâche globale à laquelle chaque entité contribue par ses calculs locaux et les communications qu'elle entreprend, sans même qu'il ait connaissance de l'état global du système. La principale motivation d'un système distribué est la nécessité d'un partage des ressources. Les ressources peuvent être des services, données, matériels, systèmes sur le web.... Cependant, l'exploitation des systèmes distribués ainsi que l'utilisation d'applications distribuées posent des défis algorithmiques fondamentaux liés aux caractéristiques et aux contraintes des ces systèmes. Dans ce qui suit est donnée une liste de problèmes distribués organisée en deux sous-sections: Problématiques générales et quelques exemples de problèmes distribués.

2.4.1 Problématiques générales

Nous présentons maintenant une liste de problèmes généraux de la mise en œuvre des systèmes distribués.

2.4.1.1 Mise à l'échelle (*scalability)*

La mise à l'échelle est la capacité d'un système de maintenir son comportement correct et la vitesse souhaitée, indépendamment du nombre d'utilisateurs. Pour ce faire il est nécessaire :

- Concevoir le logiciel de sorte que l'augmentation des utilisateurs ne nécessite de changements majeurs.
- Éviter les algorithmes et structures de données centralisés (utiliser la réplication de données si nécessaire).
- Contrôler l'augmentation des coûts due à l'augmentation des ressources disponibles.
- Le contrôle de la perte de performances (services de réplication).

2.4.1.2 Système ouvert (*openness*)

L'ouverture d'un système est la capacité du système d'être extensible, que ce soit en matériel ou en logiciel. Des nouveaux composants doivent être ajoutés sans compromettre le fonctionnement des composants existants et doivent pouvoir communiquer entre eux. Pour cela, il est important que :

- Les interfaces des nouveaux composants doivent être connues par la publication de leur documentation.
- Des protocoles et des formats standards soient utilisés .

2.4.1.3 La répartition de ressources

Un des intérêts principaux d'un système distribué est la répartition de ressources : permettre aux processeurs un accès exclusif et équitable aux ressources qu'ils demandent est un problème crucial des systèmes distribués. Un exemple représentatif du problème de la répartition de ressources est l'exclusion mutuelle de plusieurs ordinateurs partageant une ressource commune et unique dont l'accès ne peut se faire que par un seul processus à la fois. L'exclusion mutuelle doit aussi garantir l'accès à cette ressource par tous les processus. Les ressources doivent être utilisées par un seul processus à chaque fois.

2.4.1.4 Transparence

Un système distribué est un système qui s'exécute sur un ensemble de machines autonomes, mais que l'utilisateur voit comme une seule et unique machine, i.e., pour l'utilisateur la collection de machines distribuées doit être transparente. Dans le standard de Traitement Distribué Ouvert (*Open Distributed Processing, ODP*) les types de transparence sont les suivantes :

- Transparence d'accès : permet que l'accès aux ressources locales et aux ressources éloignées soit fait par les mêmes opérations (i.e., en utilisant la même interface).
- Transparence de localisation : permet aux ressources d'être accessibles sans connaissance de leur emplacement, par exemple, les programmes de courriel électroniques.
- Transparence de concurrence : permet l'exécution simultanée de plusieurs opérations sur le même ensemble de ressources sans provoquer d'interférences entre elles.
- Transparence de réplication : possibilité de dupliquer certains éléments/ressources pour augmenter la fiabilité. Permet d'utiliser plusieurs instances de la même ressource sans connaissance des répliques par les utilisateurs.
- Transparence des fautes : permet au système de fonctionner en présence de défaillances matérielles ou logicielles sans qu'un utilisateur s'interrompe (ou même se rende compte) à cause d'une faute d'une ressource. Autrement dit, une faute doit être corrigée sans que personne (ni un utilisateur, ni même parfois le système) ne s'aperçoive qu'une faute a eu lieu. C'est une propriété qui est utilisée pour les algorithmes autostabilisants: il ne s'agit pas d'une procédure de récupération sur erreur détectée, mais de correction de l'erreur par l'algorithme lui-même.
- Transparence de mobilité : permet aux ressources changer leur emplacement sans affecter leur utilisation, par exemple, téléphones portables en mouvement.
- Transparence de performance : possibilité de reconfigurer le système pour augmenter les performances sans que les utilisateurs soient au courant.
- Transparence d'échelle : permet l'expansion du système et de ses applications sans nécessiter d'importants changements de l'infrastructure existante.

Dans un système distribué les deux plus importants types de transparence sont l'accès et la localisation. Leur présence (ou leur absence) affectent profondément l'utilisation des ressources dans un système distribué. Ces deux ensembles sont dénommés *transparence de réseau*.

2.4.1.5 La synchronisation

La synchronisation est l'action de coordonner plusieurs opérations entre elles en fonction du temps. Dans un système centralisé il existe une horloge globale à laquelle tous les processeurs ont accès. Contrairement, dans un système distribué il y a plusieurs horloges non synchronisées, i.e., le système est asynchrone. La conception d'algorithmes sur systèmes synchrones est souvent plus facile que sur systèmes asynchrones. Dans les systèmes synchrones tous les processus exécutent une action en même temps et ils sont tous commandés par une horloge commune, générée par le système. Dans les systèmes asynchrones, chaque processus fonctionne à sa propre vitesse. L'ordre d'exécution des actions successives ne peut donc être déterminé à l'avance. Le type d'algorithme présenté en [49] permet d'implémenter des algorithmes synchrones dans un système asynchrone et assurer que les actions de l'algorithme synchrone sont exécutées par phases.

2.4.2 Exemples de problèmes distribués

Dans ce qui suit est donnée une liste non exhaustive d'exemples de problèmes distribués.

2.4.2.1 Sécurité

La sécurité de ressources informatiques est maintenue lorsque :

- Le niveau de confidentialité est maintenu, i.e., protection contre les accès non autorisés.
- La protection contre l'altération ou la corruption de données ou des programmes est assurée.
- La disponibilité du système est maintenue, i.e., protection contre les interférences avec les moyens d'accès aux ressources.

2.4.2.2 Le routage

Le routage est le mécanisme permettant d'acheminer des données d'un processeur à un autre, même s'ils ne sont pas voisins. Deux méthodes classiques sont utilisées pour résoudre ce problème : la diffusion [50] et la gestion de tables de routage [51].

2.4.2.3 L'élection du leader

Les systèmes distribués permettent à plusieurs processeurs se trouvant dans des endroits différents, de travailler ensemble pour accomplir une tâche commune, ce qui permet le partage de ressources et la répartition de services. Toutefois, en raison de la difficulté d'obtenir une vision globale du système, le développement d'algorithmes distribués est très complexe. Plusieurs

applications distribuées font usage d'un processeur particulier pour exécuter une tâche particulière. Ce processeur est généralement appelé *leader* et aura un comportement différent des autres dans le algorithme. Le problème d'élection du leader consiste à partir d'une configuration où tous les processus sont candidats, à atteindre une configuration où un processus est déclaré leader et tous les autres sont déclarés *battus*. Cependant, pour certains problèmes, il peut être nécessaire de distinguer au cours d'une exécution un nouveau processeur parmi tous les processeurs du réseau. Donc, les algorithmes distribués doivent pouvoir changer le rôle d'un ou plusieurs composants du système. Nombreux algorithmes ou protocoles d'élection du leader pour différentes topologies sont disponibles dans la littérature [52][53].

2.4.2.4 Le calcul d'un état global

Dans un système distribué, la séparation physique empêche un nœud de connaitre l'état global du système de manière instantanée. Chaque nœud d'un système distribué connaît son état et celui de ses voisins, donc l'état global d'un système distribué est constitué de l'ensemble des états locaux (i.e., l'état local d'un processus P_i résulte de son état initial et de la séquence d'événements dont ce processus a été le siège) des processeurs et des canaux de communications qui le constituent, pris à des instants différents mais de manière à rendre une information utile sur l'état du système dans son intégralité. La détection d'un état global d'un système distribué est illustrée dans [54]. L'algorithme de détection d'un état global peut être illustré, par exemple, par un groupe de photographes qui observent une scène dynamique, le vol d'oiseaux migrateurs. L'algorithme joue le rôle des photographes. La scène est si vaste qu'elle ne peut être prise par un seul photographe. Différents photographes doivent donc se charger de prendre chacun une partie de la scène tout en sachant que les photographies ne peuvent être prises au même instant et en exigeant que les photographes ne perturbent pas le phénomène à observer. Il faut aussi que la scène reconstituée ait « un sens », il reste à définir ce qu'est avoir *un sens* et déterminer la manière de procéder pour faire ces prises de vue.

2.4.2.5 Détection de la terminaison

La détection de la fin d'un algorithme distribué n'est pas un problème trivial. Plusieurs processus coopèrent à la réalisation d'une tâche commune et communiquent par messages. La réception d'un message par un processus peut déclencher une nouvelle phase de calcul. Pour détecter la terminaison, il ne suffit pas d'observer que l'ensemble des processeurs sont au repos ou passifs : des processeurs ont pu, avant de s'arrêter, émettre des messages qui provoqueront la reprise de l'activité. La terminaison ne peut donc être garantie que si tous les processeurs sont arrêtés et qu'il n'y a plus de message en transit. Nous pouvons noter que le problème de la terminaison se rapproche de celui de *l'interblocage* (deadlock), en ce sens qu'il s'agit, dans les deux cas, de propriétés stables : un ensemble de processeurs interbloqués ou un ensemble de processeurs coopérants terminés sont dans un état qui n'évoluera plus au cours du temps. Il y a néanmoins des différences importantes : en

particulier, l'ensemble des processeurs dont on veut détecter la terminaison est donnée initialement, alors que l'ensemble des processeurs qui risquent un interblocage n'est généralement pas connu à priori. Un algorithme de détection de la terminaison utilisant un jeton est présenté en [55] et décrit dans ce qui suit.

Soit un ensemble de sites organisés en anneau virtuel, avec un processeur par site. L'anneau est le seul moyen de communication entre les sites et on suppose que les messages ne peuvent se doubler sur l'anneau. Si chacun des sites a été visité deux fois de suite par le jeton et qu'il est resté passif entre les deux passages, nous pouvons affirmer que la terminaison est détectée. En effet, les messages qui pouvaient être en transit vers un site lors de la première visite du jeton lui sont nécessairement parvenus lors de la seconde visite, et il ne peut plus y en avoir en transit, puisque les autres sites ont été observés passifs et que les messages ne peuvent se doubler. La terminaison, si elle se produit, est ainsi détectée au plus en deux tours de jeton.

Pour vérifier qu'un processeur est resté inactif entre deux passages du jeton, on associe à chaque site une couleur (par exemple, blanc ou noir). Le site qui possède le jeton ne le renvoie que lorsque le processeur du site est passif, le site est alors mis à blanc. Si le processeur devient actif, le site est mis à noir. Lorsqu'un site blanc reçoit le jeton, nous pouvons affirmer que le processeur est resté passif en permanence depuis le dernier passage. Le jeton est muni d'un compteur qui enregistre le nombre total de sites trouvés blancs, ce compteur est remis à zéro si un site est trouvé noir. La terminaison est détectée lorsque la valeur du compteur est égale au nombre total de sites.

2.5 Modélisation d'un système distribué

Pour pouvoir effectuer une analyse du système et trouver des solutions aux problèmes qui s'y posent, il faut représenter le système par un modèle. Ce rôle est souvent pris par un graphe, non orienté, connexe et simple. Dans la Figure 2.2 nous pouvons voir un système distribué, représenté par un graphe où:

- les sommets représentent les processus
- les arêtes représentent les canaux de communication
- l'algorithme distribué est local : algorithme qui s'exécute sur chaque sommet (en utilisant uniquement le contexte local).

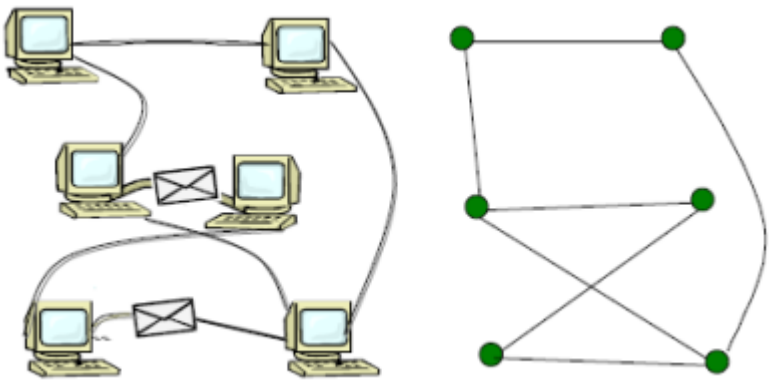


Figura 2.2 – Représentation d'un système distribué par un graphe

En fonction de la complexité du problème à résoudre les systèmes distribués, peuvent également être représentés par des chaînes, étoiles, anneaux, grilles, arbres ou cliques où les arêtes représentent les liens de communication (ou canaux) et les sommets les processus. La Figure 2.3 montre chacune de ces topologies. Ces canaux de communication peuvent être soit unidirectionnels, soit bidirectionnels, c'est-à-dire les processus échangent des données dans un seul sens ou dans les deux sens. Ainsi si un processus P_1 envoie des données à P_2 par un canal unidirectionnel, celui-ci ne pourra pas envoyer d'informations à P_1 par le même canal. Avec des liens unidirectionnels, il existe des topologies particulières telles que l'anneau unidirectionnel ou plus généralement le graphe fortement connexe (voir Figure 2.4).

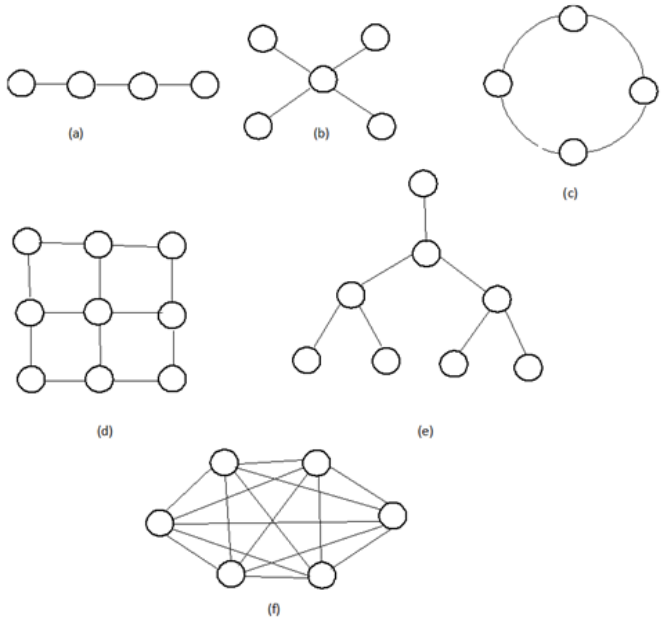


Figure 2.3 – Topologies non orientées usuelles : (a) la chaîne, (b) l'étoile, (c) l'anneau, (d) la grille, (e) l'arbre, (f) le clique

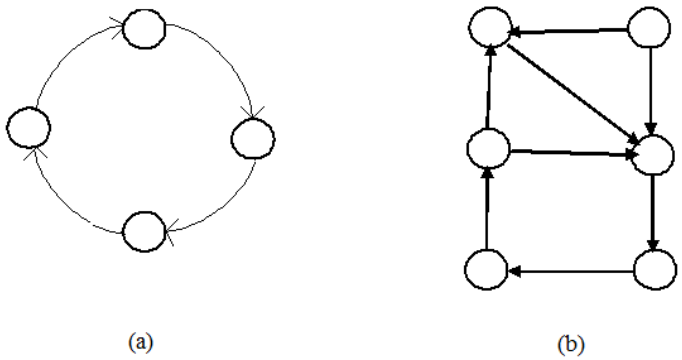


Figure 2.4 - Topologies orientées usuelles : (a) l'anneau unidirectionnel, (b) graphe fortement connexe

Dans les canaux de communication, il existe plusieurs hypothèses possibles pour les communications, de la communication globale à la communication point à point, en passant par la communication multi-points. Dans une communication globale, un processus peut communiquer avec tous les autres. Dans une communication multi-points, un sous-ensemble de processus peuvent communiquer ensemble alors que dans la communication point-à-point, seulement deux processus peuvent communiquer à travers les liens qui les relient.

Un algorithme ou programme se compose d'un ensemble de processus. Un processus contient un ensemble de *constantes* qu'il peut lire mais qu'il ne peut pas mettre à jour. Un processus maintient un ensemble de *variables*. Chaque variable s'étend sur un domaine fixe de valeurs. Des lettres minuscules peuvent être utilisées pour indiquer des variables seules, et celles majuscules pour désigner des ensembles. Certaines variables sont persistantes d'une activation du processus à l'autre, et sont appelées *variables d'état*, alors que d'autres sont éliminées entre deux activations d'un processus, et sont appelés *variables locales*. Quand il n'y a pas d'ambiguïté, la variable de terme générique se réfère à une variable d'état.

Une action a la forme (*nom*) : (*garde*) → (*commande*) [43]. Un *garde* est un prédicat booléen. Dans le modèle de mémoire partagée, ce prédicat est au-dessus des variables du processus et de ses voisins de communication. Dans le modèle de registre partagé, ce prédicat est sur les variables du processus et un registre d'entrée unique. Dans le modèle de passage de messages, ce prédicat est sur les variables du processus et la primitive *recevoir(m)*. Dans ce contexte, *recevoir(m)* est évalué à *true* si un message correspondant à *m* a été reçu, et à *false* sinon. Une *commande* est une séquence d'instructions d'affectation de nouvelles valeurs aux variables du processus (dans tous les modèles de communication) et l'envoi de message (en ustilisant la primitive *envoyer(m)*) dans le modèle de

passage de message. Un *paramètre* est utilisé pour définir un ensemble des actions comme une action paramétrisée. Par exemple, soit ***j*** un paramètre allant sur les valeurs 2,5 et 9, alors une action *ac.**j*** définit l'ensemble des actions: *ac.(j:=2) [] ac.(j:=5) [] ac.(j:=9).* Où *[]* indique *l'alternance* (c'est à dire *a[]b* indique le fait que soit *a* ou *b* est exécutée, mais pas les deux, et le choix n'est pas déterministe).

Une *configuration* du système est l'affectation d'une valeur à chaque variable de chaque processus à partir du domaine de la variable correspondante (et éventuellement une affectation semblable de contenu du message dans les canaux entre les processus dans le cas du modéle de passage de message). Chaque processus contient un ensemble d'actions. Une action est *activée* dans une configuration si sa garde est *vraie* à cet état. Un *calcul* est une séquence maximale de configurations telle que pour chaque configuration s_i, la configuration suivante s_{i+1} est obtenue en exécutant la commande d'une action qui est activé en s_i.

2.6 L'autostabilisation

L'autostabilisation est une propriété d'un système distribué, composé de plusieurs unités de calcul capables de communiquer entre elles, qui lui permet, lorsque le système est mal initialisé ou perturbé, à retourner automatiquement à un fonctionnement correct en un nombre fini d'étapes de calcul. Ce concept a été introduit par Dijkstra en 1974 [44].

Un système distribué qui est autostabilisant finira dans une configuration sûre (ou légitime) indépendamment de la configuration initiale. Une configuration est dite légitime [56] quand la spécification du problème à résoudre est vérifiée. Si la spécification n'est pas vérifiée on dira que le système est dans une configuration illégitime.

La propriété de l'autostabilisation permet au système distribué tolérer des défaillances transitoires, en particulièr des défaillances que ne modifient pas le code exécuté par n'importe quel nœud. Après une défaillance ou une série de défaillances, le système atteint dans un temps fini une configuration arbitraire (c'est à dire une configuration où les variables ne sont pas initialisées), et finira par récupérer un comportement correct si aucune autre défaillance se produit. L'autostabilisation ne nécessite de garanties probabilistes pour affirmer que le système est tolérant aux fautes. Vu que le système se stabilise indépendamment de l'état initial, nous pouvons affirmer que le système autostabilisant est parfaitement tolérant aux fautes.

L'autostabilisation vise des applications dans les domaines où l'intervention d'un humain pour rétablir le système après une défaillance est impossible ou dans lesquelles il est préférable de s'en

passer: A titre d'exemple peuvent être mentionnés les réseaux informatiques, les réseaux de capteurs et les systèmes critiques (applications avioniques, satellites,….).

Par conséquent, l'autostabilisation est une combinaison des deux propriétés :

- Propriété de ***clôture*** : elle assure que partant d'une configuration légitime et sans l'occurrence d'une faute, le système restera dans une configuration légitime.
- Propriété de ***convergence*** : à partir d'une configuration arbitraire le système atteint en un temps fini une configuration légitime.

Ces propriétés garantissent que le système retrouvera un comportement normal à partir d'une configuration quelconque. La Figure 2.5 illustre le comportement d'un système autostabilisant.

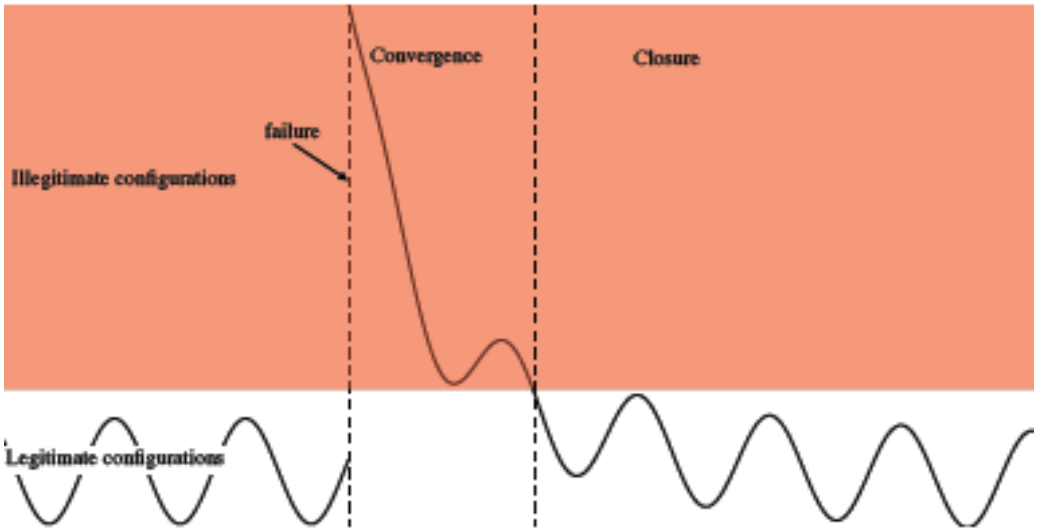


Figure 2.5 - Comportement d'un système autostabilisant

Un algorithme auto-convergent qui vérifie ces deux propriétés est adapté à tolérer les fautes transitoires. En effet, après une défaillance transitoire les composants (mémoire ou liens de communications) peuvent contenir des informations erronées, mais si l'algorithme est autostabilisant, cela devrait converger dans un temps fini vers un état légitime.

2.6.1 L'autostabilisation en pratique

L'autostabilisation assure de retrouver automatiquement un état légitime du système, c'est à dire un fonctionnement correct, après un certain temps de fonctionnement sans qu'une nouvelle défaillance se produise. En particulier, cela permet de tolérer toute défaillance transitoire, modification arbitraire de l'état d'un processus en cours d'une exécution. Une telle défaillance peut être causée, par exemple, par un rayon cosmique frappant un circuit intégré. Elle peut aussi être due au fonctionnement du système dans de mauvaises conditions, en particulier en cas de surchauffe ou de sur-cadencement

(ou *overclocking)*, qu'est une manipulation ayant pour but d'augmenter la fréquence du signal d'horloge d'un processeur au-delà de la fréquence nominale afin d'augmenter les performances de l'ordinateur. Une défaillance transitoire peut laisser le système dans une configuration quelconque, totalement imprévisible. Le retour automatique à une configuration légitime est souhaitable dans un système sur lequel il n'est pas possible de faire intervenir un technicien, par exemple dans un satellite.

La garantie d'autostabilisation peut à première vue ne pas paraître aussi prometteuse que celle d'algorithmes de tolérance aux fautes traditionnels, qui visent à garantir que le système reste toujours dans un état légitime sous certains types de transitions d'état. Cependant, cette condition ne peut pas toujours être atteinte. Par exemple, lorsque le système est démarré dans un état illégitime ou est corrompu par une faute. En outre, les systèmes distribués sont difficiles à déboguer, en raison de leur complexité. Par conséquent, il est difficile d'empêcher un système distribué d'atteindre un état illégitime. Il est important de noter la généralité des algorithmes autostabilisants: contrairement à d'autres approches, ils garantissent le fonctionnement quelque soit le type d'erreur transitoire, sans avoir besoin de détailler les causes et les effets de chaque type d'erreur.

2.7 L'algorithme auto-convergeant

L'autoconvergence est la propriété d'un système distribué, composé de plusieurs processus capables de communiquer entre eux, qui consiste, lorsque le système est mal initialisé ou perturbé (c'est à dire se trouve dans un état illégitime ou illégale), à converger automatiquement à un état légitime ou légale (l'état où le fonctionnement est correct) en un nombre fini d'étapes de calcul. Cette approche contraste avec l'avis typique d'un algorithme de tolérance aux fautes qui assure que le système ne sortira jamais d'un état correct. L'autostabilisation vise des applications dans les domaines où l'intervention d'un humain pour rétablir le système après une défaillance est impossible ou dans lesquels il est préférable de s'en passer: Á titre d'exemple peuvent être mentionnés les réseaux informatiques [57][58], les réseaux de capteurs [59-69], le plus court chemin [70][71] et les systèmes critiques (applications avioniques, satellites,….). Cette capacité à se remettre sans intervention extérieure est hautement souhaitable, car elle permet de réparer les erreurs et revenir à un fonctionnement normal pour son propre compte. Dans ce contexte d'autostabilisation on peut dire que les algorithmes auto-convergeants n'ont pas besoin d'être reinitialisés et ils peuvent récupérer un état légitime suite à l'occurrence de fautes transitoires. Si l'on considère, par exemple, des réseaux informatiques il serait déraisonnable d'arrêter un réseau et le réinitialiser lors de l'ajout ou la suppression de nœuds.

L'autostabilisation est une propriété souhaitable pour les algorithmes auto-convergeants, et sa convergence vers un état correct se produit en un nombre fini d'étapes. Cependant, l'autostabilisation

ne définit pas combien de temps elle se produit, ce qui suscite des préoccupations à propos de la complexité de la convergence.

Dans cette thèse la notion d'autostabilisation sera transposée aux algorithmes centralisés, exécutés par un unique processeur, qui renvoient une valeur, où la partie expérimentale se concentre dans une application dédiée à rechercher les plus courts chemins dans un graphe.

2.7.1 Le problème du plus court chemin

Le problème du plus court chemin consiste à déterminer le chemin de coût minimum reliant deux nœuds (ou sommets). Un exemple est de trouver le chemin le plus rapide pour aller d'un endroit à un autre sur une carte routière; dans ce cas, les sommets représentent les emplacements et les arcs représentent les segments de route et sont pondérés par le temps nécessaire pour parcourir ce segment. La Figure 2.6 montre un graphe avec 6 nœuds et 7 arcs, dans cette figure (6, 4, 5, 1) et (6, 4, 3, 2, 1) sont deux chemins entre les sommets 1 et 6.

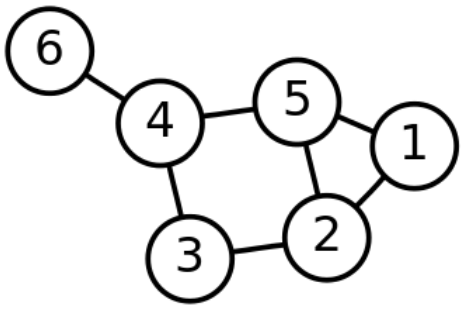


Figure 2.6 - Un graphe avec 6 nœuds et 7 arcs

Problème:

- Soit un graphe G = (V, A) pondéré défini par sa matrice (un tableau) et sa taille (un entier). Où *V* est l'ensemble des nœuds ou sommets et *A* est l'ensemble des arcs (ou arêtes).
- Un coût t_{ij} est associé à chaque arc (i,j) $\in$A. Ceci peut être: une distance, un temps de trajet, etc.
- Soit le nœud *0*, le nœud correspondant au point final.
- Nous cherchons le chemin de coût minimum (ou plus court chemin) reliant n'importe quel nœud à le nœud *0*.

2.7.2 Fonctionnement de l'algorithme

Le but de programme appelé PCCAS (Plus Courts Chemins Auto Stabilisants), est la recherche des plus courts chemins dans un graphe. Le graphe est représenté par une matrice d'entiers

T de taille *N* x *N* avec des valeurs prédéfinies. Les arêtes que relient les nœuds *i* et *j* ont une longueur T_{ij}. La sortie est représentée par une matrice *D* de taille *N* x *1* dont les éléments ne sont pas prédéfinis. Les variables *b* et *c* sont utilisées pour détecter la convergence de l'algorithme, i.e., elles indiquent que le calcul est terminé. Les entiers *i* et *j* sont les compteurs de boucle nécessaires pour effectuer les calculs matriciels, i.e., ils parcourent les sommets du graphe.

Le code C de l'algorithme auto-convergeant est donné ci-dessous :

```
b = c = 1
T = N × N matrix
D = N × 1 matrix
while (b || c){
   c = b;
   b = 0;
   D[0] = 0;
   for(i = 1; i < N; i++){
      m = INFINIE ;
      for( j = 0; j < N; j++){
         if (m >= D[j] + T[N * i + j])
            m = D[j] + T[N * i + j];
      }
      if (D[i] != m)
         b = 1;
      D[i] = m;
   }
```

L'algorithme ci-dessus est un benchmark[1] dont des détails peuvent être trouvées dans les références [72-75]. La solution pour ce programme se résume à choisir le nœud non visité avec la distance plus faible, calculer la distance entre lui et chaque nœud voisin non visité, et mettre à jour la distance de chaque voisin si cette distance est la plus petite.

La matrice en (2.1) représente le graphe dans lequel deux nœuds *i* et *j* sont reliés par une arête de longueur T_{ij} :

[1]Algorithme développé par l'équipe CaRO (Calcul Réparti et Optimisation) du Laboratoire PRiSM (Parallélisme, Réseaux, Systèmes, Modélisation), Université de Versailles Saint-Quentin en Yvelines.

$$T_{N\times N} = \left[\begin{array}{l} \underbrace{\overbrace{T_{00}}^{j=0}\ \overbrace{T_{01}}^{j=1}\ \overbrace{T_{02}}^{j=2}\cdots\cdots\ \overbrace{T_{0N}}^{j=N}}_{i=0} \\ \underbrace{\overbrace{T_{10}}^{j=0}\ \overbrace{T_{11}}^{j=1}\ \overbrace{T_{12}}^{j=2}\cdots\cdots\ \overbrace{T_{1N}}^{j=N}}_{i=1} \\ \underbrace{\overbrace{T_{20}}^{j=0}\ \overbrace{T_{21}}^{j=1}\ \overbrace{T_{22}}^{j=2}\cdots\cdots\ \overbrace{T_{2N}}^{j=N}}_{i=2} \\ \vdots \\ \vdots \\ \underbrace{\overbrace{T_{N0}}^{j=0}\ \overbrace{T_{N1}}^{j=1}\ \overbrace{T_{N2}}^{j=2}\cdots\cdots\ \overbrace{T_{NN}}^{j=N}}_{i=N} \end{array} \right] \qquad (2.1)$$

À chaque itération, on considère chaque nœud *i* et l'on calcule *m* à min $\{D_j + T_{ij}\}$. $D_j + T_{ij}$ représente la longueur du plus court chemin pour aller de *i* à *0* en passant par *j* et le plus court chemin connu de *j* à *0*. Alors, la distance entre un nœud *i* et le nœud *0* (nœud de référence) est la seule quantité qui satisfait la relation :

$$\begin{aligned} &D_0 = 0; \\ &D_i = \min\{D_j + T_{ij}\} \end{aligned} \qquad (2.2)$$

Le programme PCCAS établit D_i à $\min\{D_j+T_{ij}\}$ pour tout *i* jusqu'à ce qu'il converge vers la solution, i.e., seule la matrice *D* vérifie la relation (2.2). L'équation (2.3) ci-dessous représente le calcul de la matrice *D* :

$$D = \left[\sum_{i=1}^{N} \left(\sum_{j=0}^{N} \underbrace{\left[D_j + T_{ij} \right]}_{m} \right) \right] \qquad (2.3)$$

Dans ce qui suit est donné un résumé de la solution de PCCAS : Les valeurs choisies pour la matrice d'entrée *T* (2.1) sont engendrées aléatoirement (voir l'ANNEXE A). Les variables *b* et *c* sont initialisés à *1*. La matrice *D* et la variable *m* ne sont pas initialisées. En effet, le nœud *0* est toujours à distance *0* de lui-même, et l'algorithme commence toute itération en mettant *D[0]* à *0* (ligne 7 d'algorithme). Ensuite dans la boucle *while* la variable *c* prend la valeur qu'avait *b* à l'itération précédente. Dans la boucle *for* sur *i* : *m = INFINIE,* sert à indiquer qu'aucun chemin avec distance minimale entre *i* et *0* n'a pas encore été trouvé. Dans la boucle *for* sur *j*: *D[j] +T[N*i+j],* la longueur du plus court chemin pour aller de *i* à *0*, est calculée à chaque itération pour chaque nœud *i,* c'est à dire l'on calcule *m = min* $\{T_{ij} + D_j\}$ (ligne 11 de l'algorithme). Si la condition dans la ligne 11 n'est pas vérifiée la variable *m* reste avec la même valeur que celle qu'elle avait dans le calcul précédent. Après sortie de la boucle *for* sur *j*, la condition *D[i] != m* (ligne 14 de l'algorithme) est vérifiée pour chaque

nœud à chaque itération, si elle est vrai ceci signifie que l'algorithme n'a pas convergé et donc la variable *b* est assignée "vrai" (*b = 1*) et il faudra donc faire des itérations jusqu'à ce qu'il converge vers une solution, c'est à dire à la seule matrice *D* qui satisfait l'équation (2.2). Cette solution est obtenue lorsque au cours des itérations successives la matrice *D* a toujours les mêmes résultats, ainsi la variable *b* passe à "faux", et dans l'itération suivante *c* (*c = b* dans la ligne 5 d'algorithme) passe a "faux", et l'on sort de la boucle *while* conduisant à la fin d'algorithme. Comme dit précédemment le PCCAS est une application dédiée à rechercher les plus courts chemins dans un graphe. N'importe quelle erreur peut se produire au cours du calcul, si elle n'entraîne pas la fin du programme, la réponse du programme sera correcte. Autrement dit, si les variables sont perturbées ou pas, l'auto-convergence est vérifiée de la même façon, c'est à dire, une solution auto-convegeante consiste à vérifier pour chaque nœud si les sommes de ses distances à ses voisins et la longueur de leurs plus courts chemins à *0* sont minimum. Si cette condition est vérifiée pour tous les nœuds, les distances sont exactes et on arrive au plus court chemin pour aller de *i* à *0*. Un exemple de calcul des plus courts chemins dans un graphe est donné dans l'ANNEXE B.

2.7.3 Problèmes potentiels suite à l'occurrence d'un SEU

Après la présentation du fonctionnement du programme PCCAS, supposons maintenant que les données ou les variables de ce programme soient corrompues. La corruption de *D* ou de *m* est prise en charge par l'algorithme (ils ne sont pas initialisés). En effet, le nœud *0* est toujours à distance *0* de lui-même, voir l'équation (2.2) ci-dessus. À l'étape suivante un SEU pertubant un registre ou la mémoire, cette erreur va se propager ou être corrigée, mais l'itération ne s'arrêtera que lorsque seront vérifiées des conditions de cohérence locale à chaque nœud c'est-à-dire: pour chaque nœud la longueur de son plus court chemin à *0* doit être le minimum des sommes de ses distances à ses voisins et la longueur de leurs plus courts chemins à *0*. Si cette condition est vérifiée pour tous les nœuds, alors il est sûr que les distances sont exactes.

Si le graphe ou la taille du graphe, respectivement *T* et *N*, sont corrompus, ces fautes altèreront certainement les résultats, puisqu'il s'agit des données d'entrée, c'est-à-dire que l'algorithme aboutira aux plus courts chemins sur les données corrompues.

Si le booléen ***b*** ou les compteurs de boucle sont corrompus, alors les résultats peuvent être erronés, suivant la corruption. Si ***b*** passe à faux à la fin de l'itération, on peut en sortir indûment. De même, si au début de l'itération, ***i*** passe à une valeur supérieure à ***N***, on risque de manquer des modifications, donc de laisser ***b*** passer à la valeur « faux » alors qu'il devrait passer à « vrai ». Les mêmes remarques sont valables pour ***j***.

Dans l'hypothèse d'une erreur isolée, une redondance de ***b*** a été introduite pour que l'algorithme s'arrête. Cette redondance (variable booléene ***c***) prend la valeur qu'avait ***b*** à l'itération précédente, donc l'algorithme s'arrêtera après deux itérations sans modification.

En principe, dans l'hypothèse d'une erreur isolée, l'algorithme devrait donc toujours converger vers le bon résultat, les seules erreurs qui peuvent conduit à des résultats erronés sont :

- Les erreurs qui font sortir de la boucle avant de sa fin : la valeur de ***D*** calculée dans la boucle *while* précédente est exactement la même que la nouvelle ***D***, quand les compteurs de boucles ***i*** et/ou ***j*** passent à des valeurs supérieures à ***N*** de leurs boucles respectives et quand les variables *b* et *c* sont égales à *0* conduisant à *while (b||c) =0*.

- Les erreurs affectant le prochain ***D***, qui se produisent après la fin de la boucle *while*, ou lors de la dernière itération sur un élément de l'index $j_0 < j$.

Dans le cas des erreurs multiples, si plusieurs variables ou données sont perturbées simultanément, l'algorithme converge pour peu que les erreurs ne soient pas :

- ***b*** et ***c*** passant à faux de façon erronée dans le temps d'une itération.

- ***c*** passant à faux, et ***i*** ou ***j*** passant à une valeur supérieure à ***N*** dans le temps d'une itération (cette deuxième erreur n'aboutit d'ailleurs pas nécessairement à un résultat faux).

Pour faire face aux erreurs, nous pouvons exiger qu'aucun changement ne se produise dans deux itérations successives de la boucle *while* avant de la quitter. Puis, á l'aide de deux variables booléennes ayant le même rôle, nous nous assurons qu'une seule erreur, sauf celles se produisant sur le compteur de programme, ne peut en principe conduire à un résultat incorrect.

2.8 Conclusion

Dans ce chapitre ont été décrits les systèmes distribués présentant leurs caractéristiques générales ainsi que les défis de la mise en œuvre de ces systèmes. Nous avons donc présenté l'approche d'autostabilisation qui permet de rendre tolérant aux fautes un système distribué. Cette approche sera au cœur de l'algorithme d'auto-convergence utilisé le long de cette thèse. Les potentiels problèmes de cet algorithme ont été montrés. Dans la suite de ce manuscrit sera décrit l'algorithme auto-convergeant, présenté dans ce chapitre, exécuté par un processeur LEON3 implémenté dans un FPGA (Field Programmable Gate Array). Des modifications nécessaires pour implémenter des

techniques de tolérance aux fautes seront effectuées sur l'algorithme. La robustesse face aux SEU de l’algorithme modifié sera évaluée par des campagnes d'injections de fautes.

Chapitre 3. Étude par injection de fautes de la robustesse d'un algorithme auto-convergeant

Ce chapitre présente une étude par injection de fautes de la robustesse d'un algorithme (c'est un « cas d'étude ») d'auto-convergence. Cet algorithme a été exécuté par un processeur LEON3 implémenté dans un FPGA embarqué dans une plateforme de test spécifique. Des fautes de type SEU (Single Event Upset) ont été injectées en utilisant la méthode CEU (Code Emulated Upset). Les résultats obtenus dans les campagnes d'injection de fautes effectuées ont mis en évidence une certaine sensibilité aux SEUs de la première version de l'algorithme étudié. Une version améliorée de l'algorithme d'auto-convergence, incluant des techniques de tolérance aux fautes au niveau logiciel, a été développée et testée par injection de fautes ainsi comme l'analyse des fautes qui ne sont pas tolérées. À la fin de ce chapitre une version de l'algorithme d'auto-convergence avec différentes stratégies de tolérance aux fautes additionnelles (modifications logicielles et l'implémentation d'un TMR implémenté dans un processeur LEON3 trois *cores*) est présentée et évaluée par des campagnes d'injection de fautes dont les résultats mettent en évidence sa robustesse face aux SEUs.

3.1 Approche d'injection de fautes: la méthode CEU

Pour émuler l'occurrence de fautes de type SEU pendant l'exécution de l'algorithme d'auto-convergence, nous avons utilisé la méthode CEU (Code Emulated Upsets). Cette méthode a été développée à TIMA pour estimer le taux d'erreurs des divers processeurs ou microprocesseurs exécutant une application donnée (le microcontrôleur 80C51 d'intel [76] et les PowerPC 7447A et 7448 [77]), et pour valider diverses techniques de tolérance aux fautes, en particulier celles appliquées au niveau *software* [38].

La méthode CEU (Code Emulated Upsets), présentée pour la première fois en 2000 [78], est une approche consacrée aux systèmes de type processeur or microprocesseur (version hardware ou implémentés dans des FPGA). Cette méthode peut être utilisée pour effectuer des expériences d'injection de fautes au cours desquelles des SEUs sont injectées dans les ressources (registres, zones mémoire,…) du circuit testé qui sont potentiellement sensibles aux SEU et qui sont accessibles à travers le jeu d'instructions du circuit considéré.

La méthode est basée sur l'injection de *bit-flips*, à un instant et emplacement aléatoirement choisis via l'assertion de signaux d'interruptions asynchrones, signaux qui sont disponibles dans la plupart des circuits de type processeur. L'exécution du code d'interruption associé permet la modification du contenu d'un bit aléatoirement sélectionnée dans la zone sensible accessible à travers de jeu d'instructions du processeur étudié. Une fois la routine d'interruption exécutée, le processeur reprend l'exécution du programme, ce qui permet de simuler de manière assez réaliste l'occurrence d'un SEU. Parmi les ressources ciblées peuvent être mentionnés les registres généraux, les registres de fonctions spéciales, les mémoires SRAM internes, les mémoires cache, etc.

L'injection d'un SEU dans un registre d'usage général ou dans des zones de mémoire interne ou externe directement adressable requière quelques instructions pour effectuer les tâches suivantes:

- Lire le contenu de la cible sélectionnée.
- Effectuer l'opération *XOR* (ou OU exclusif) avec une valeur de masque appropriée (XOR avec un "1" à l'emplacement du bit qui va être inversé et des "0" ailleurs).
- Écriture de la valeur corrompue dans la cible.

La plupart des circuits de type processeur ont des jeux d'instructions qui permettent d'effectuer ces tâches en quelques instructions et qui nécessitent quelques cycles d'horloge. Cependant, l'aspect crucial pour émuler le basculement de contenu d'un bit est l'inclusion de ces codes dans le programme initial, pour qu'il soit exécuté d'une manière non intrusive (seulement le bit cible d'une cellule mémoire doit être perturbé) au moment voulu lors de l'exécution de l'application. Comme dans la plupart de processeurs est disponible un mécanisme d'interruption asynchrone, l'utilisation de ce mécanisme devient une solution simple et efficace pour émuler l'occurrence d'un SEU. Le code qui va causer cette perturbation, appelé code CEU, doit se terminer par une instruction de type *retour de l'interruption (RETI)*, et doit être enregistré à une adresse spécifique dans la mémoire de la plateforme de test, en tant que programme associé à l'interruption.

L'approche CEU d'injection de fautes est illustrée dans la Figure 3.1. Cette approche comporte les étapes suivantes, en réponse à l'activation d'une interruption [77] :

- Le processeur démarre l'exécution du programme.
- Activation de l'interruption à un instant choisi aléatoirement.
- Le processeur interrompt l'exécution de son programme principal après avoir terminé l'exécution de l'instruction en cours.
- Sauvegarde du contexte. Cette étape dépende strictement de l'architecture du processeur étudié. Dans la plupart des processeurs, suite à une interruption le contexte (compteur programme et certains des registres) est automatiquement sauvegardés dans une pile. Dans le cas du processeur LEON3, qu'est le processeur utilisé dans le cadre

des études faits dans cette thèse, lorsqu'un signal d'interruption est activé la fenêtre de registres (détails décrits dans la section 3.2) est changée, évitant l'utilisation d'une pile pour enregistrer le contexte qui doit être restauré après que le programme d'interruption est exécuté.

- Exécution du code CEU associé à l'interruption, provoquant le basculement du bit aléatoirement choisi.
- Retour vers le programme principal. Le processeur exécute ce programme avec le SEU injecté.
- Le processeur termine l'exécution du programme et les résultats peuvent être observés.

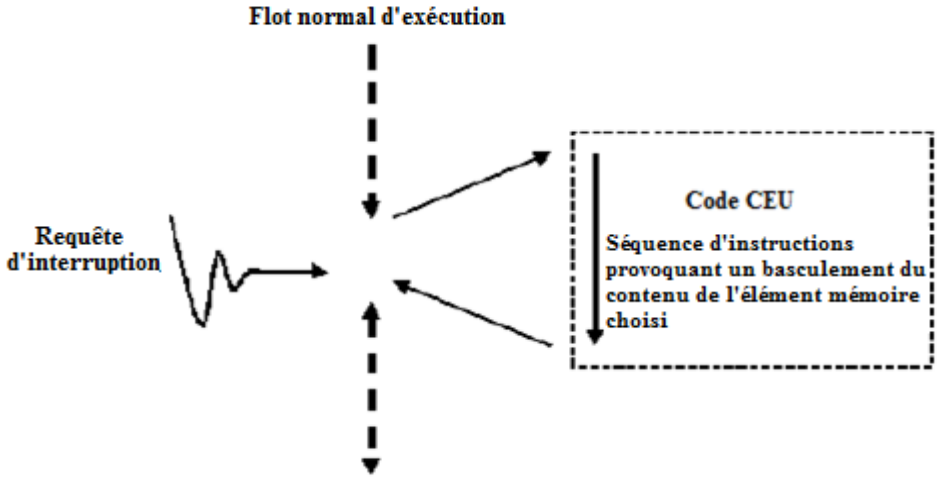


Figure 3.1 - L'approche d'injection de fautes CEU (Code Emulated Upsets)

Comme indiqué précédemment, la méthode d'injection des fautes CEU est basée sur l'activation du signal d'interruption à des instants aléatoires lors de l'exécution d'une application. Cette approche est une alternative assez réaliste pour simuler des erreurs induites sur une application exécutée par un processeur, suite à l'occurrence d'un SEU dû à l'impact des particules énergétiques [79].

Si les SEUs sont injectés durant l'exécution d'une application donnée, il peut être dérivé un taux d'erreur, τ_{inj}, qui montre le rapport entre le nombre d'erreurs détectées et le nombre de SEUs injectés:

$$\tau_{inj} = \frac{\#\text{Erreurs detectées par injections de fautes}}{\#\text{fautes injectées}} \tag{3.3}$$

Le taux d'erreur τ_{inj} peut être interprété comme le nombre moyen de SEUs nécessaires pour provoquer une erreur dans une application.

Enfin, l'approche CEU est basée sur l'injection de fautes dans des zones sensibles accessibles via le jeu d'instruction, donc cette approche ne peut pas prendre en compte le cas où une faute se produit dans les zones qui ne sont pas accessibles à travers de jeu d'instruction, comme par exemple, les registres pipeline, l'unité arithmétique et logique et les registres des machines à états.

3.2 Le véhicule de test : le processeur LEON3

L'algorithme d'auto-convergence peut être exécuté par n'importe quel processeur ou microprocesseur dans une architecture typique ou implémenté dans un FPGA et aussi par un circuit dédié. Dans ces travaux on a pris comme véhicule de test un processeur LEON3 implémenté dans un FPGA. Ce processeur a été choisie parce que c'est un processeur dont le design est Open Source, est utilisable en SMP (plusieurs processeurs) sous Linux, est basé sur l'architecture SPARC, et le design est fourni par la société Gaisler Research. Ce processeur a une architecture un peu différente des processeurs typiques surtout dans la zone concernant la fenêtre des registres. L'une des principales caractéristiques de ce processeur réside dans l'utilisation d'un large ensemble de registres organisés de sorte à qu'ils puissent être accédés par fenêtres. À tout instant, un programme voit 8 registres globaux plus une fenêtre de 24 registres (voir sous-section 3.2.1), et enfin, ce processeur a été déjà mis en œuvre dans le cadre des recherches antérieures, dans un FPGA embarqué dans une plateforme de test spécifique.

Dans une architecture typique, comme par exemple celle du microcontrôleur 8051C, où les principales caractéristiques correspondant à la famille MCS 51 sont les suivantes [80] :

- CPU 8 bits
- Processeur booléen permettant le calcul sur un bit
- Mémoire interne de 128 octets.
- 64K octets d'espace mémoire de programme.
- 64K octets d'espace mémoire des données.
- Communication bidirectionnelle UART.

Les zones sensibles aux SEUs pour un circuit typique comme le 8051C sont la mémoire RAM interne, les registres spéciaux (SFR), les registres généraux, les accumulateurs et le compteur de programme (PC). Les donnés et les variables d'un programme exécuté par ce circuit sont stockées dans la mémoire interne. Alors, si l'on considère que ce circuit exécute par exemple une multiplication de deux matrices de dimensions 6x6 qu'occupe environ 92 % de la mémoire interne du microcontrôleur la probabilité d'observer des erreurs augmente étant donné que la mémoire interne correspond à une grande partie de la zone sensible au SEU.

Le processeur LEON3 est un modèle VHDL (VHSIC Hardware Description Language) synthétisable sur des circuits numériques de type FPGA comme ceux de Xilinx ou Altera ou sous forme d'un circuit spécifique ASIC. Le modèle VHDL est hautement configurable et adaptable. Ce processeur est distribué dans le cadre de la bibliothèque GRLIB IP ce qui permet, par exemple une intégration simple dans la conception de systèmes sur puce (SoC) complexes.

Le LEON3 est un processeur 32 bits basé sur l'architecture SPARC V8 (Scalable Processor ARChitecture). Son code source complet est disponible librement sous la licence GNU, permettant l'utilisation gratuite et illimitée dans les applications de recherche. Pour les applications commerciales son code source est également disponible sous une licence commerciale à faible coût. Les principales caractéristiques de ce processeur sont les suivantes [81] :

- 7 étages de pipeline
- Jeu d'instructions SPARC V8 avec l'extension V8e.
- Haute performance et FPU (Floating Point Unit) entièrement en pipeline IEEE-754.
- Mémoires cache d'instructions et des données configurables et séparées (Harvard l'architecture)
- Interface de bus AMBA-2.0 (AHB, APB)
- Unité de gestion mémoire (MMU - Memory Management Unit).
- Cache configurable: 1 à 4 étages, 1 à 256 kbytes par étage. Remplacement aléatoire, ou LRU (Least Recently Used).
- Support multi-processeurs symétrique (SMP).
- Fonctionne jusqu'à 125 MHz sur FPGA et 400 MHz sur les technologies ASIC 0.13 μm.
- Version tolérante aux fautes, et version SEU-proof disponible pour des applications spatiales.
- Haute performance: 1.4 DMIPS/MHz, 1.8 CoreMark/MHz (gcc-4.1.2).
- Prise en charge avancée de débogage on-chip.
- Contient des multiplieurs, diviseurs et unités MAC (Media Access Control)
- Contient plusieurs blocs IP matériels: contrôleur Ethernet, unité de calcul en virgule flottante, contrôleur DMA.

La Figure 3.2 illustre les interfaces et les périphériques d'un processeur LEON3 qui est configuré pour utiliser le JTAG *debug link*, le contrôleur AHIB, le contrôleur mémoire, le bridge AHB/APB, l'UART, les TIMERS, les interruptions, et les ports I/0. Le processeur LEON3 est entièrement paramétrable grâce à l'utilisation des génériques VHDL. Dans le même design il est possible d'instancier plusieurs cœurs de processeur avec des configurations différentes. Ce processeur peut être configuré en utilisant un outil graphique construit sur *tkconfig* du noyau linux. Cette outil permet de définir rapidement une configuration adaptée sur mesure et permet encore de configurer d'autres périphériques sur puce tels que les contrôleurs de mémoire et les interfaces réseau.

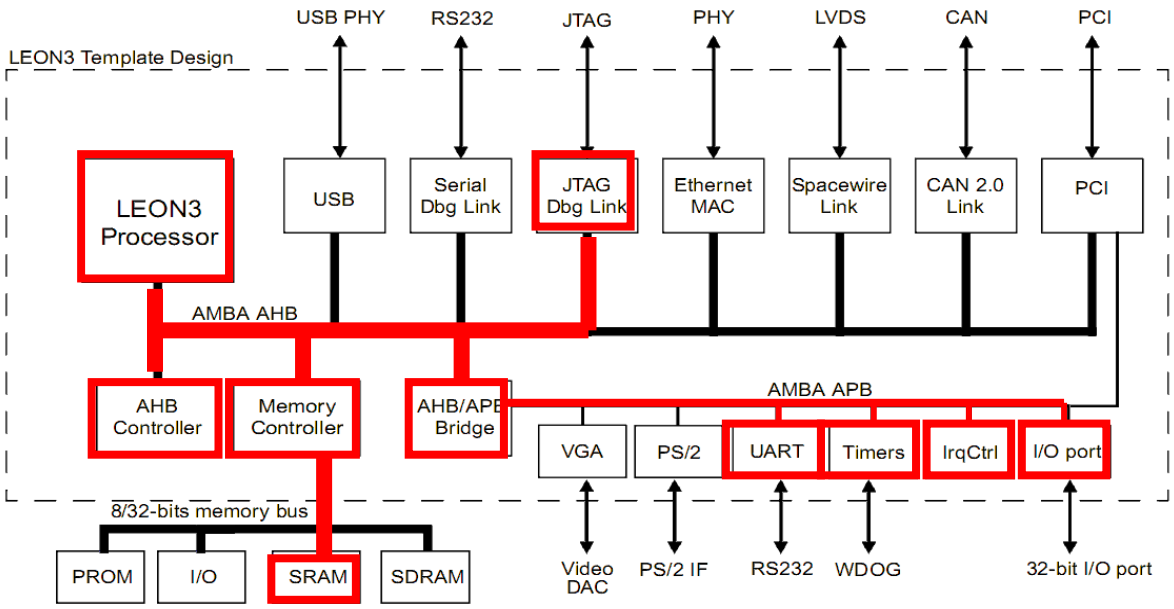


Figure 3.2 - Le processeur LEON3, les interfaces et périphériques

3.2.1 Fenêtres de registres du LEON3: Caractéristiques

L'architecture de LEON3 est organisée en fenêtres de registres, c'est à dire l'ensemble de registres sont organisés de sorte à qu'ils puissent être accédés par fenêtres. Les fenêtres de registres constituent une technique qui a pour but améliorer la performance des appels de fonctions. Elles ont été utilisées pour la première fois dans les processeurs RISC de Berkeley, qui sont á l'origine des architectures SPARC, AMD29000 et Intel i960 [82]. En d'autres mots la fenêtre de registres de cette architecture permet une réduction significative des instructions de lecture et d'écriture dans la mémoire en particulier dans programmes plus vastes. Parmi ces registres, dans les travaux faits dans cette thèse cinq d'entre eux seront utilisées à des fins d'injection de fautes et donc ne seront pas considérés comme faisant partie de la zone sensible aux SEUs (des fautes ne seront pas injectées dans les campagnes de test).

Lors de l'assertion d'un signal d'interruption ou un appel d'une fonction, la fenêtre de registres est modifiée, ce qui évite l'utilisation d'une pile pour sauvegarder le contexte, et est restaurée une fois le programme d'interruption exécuté. Dans la Figure 3.3 est donnée une représentation des fenêtres de registres de cette architecture. Parmi les jeux d'instructions de ce processeur, seulement deux instructions permettent le passage d'une fenêtre à une autre, les instructions *restore* et *save*. Le registre CWP (Current Window Pointer) est utilisé pour indiquer laquelle des 8 fenêtres est la fenêtre active du processeur.

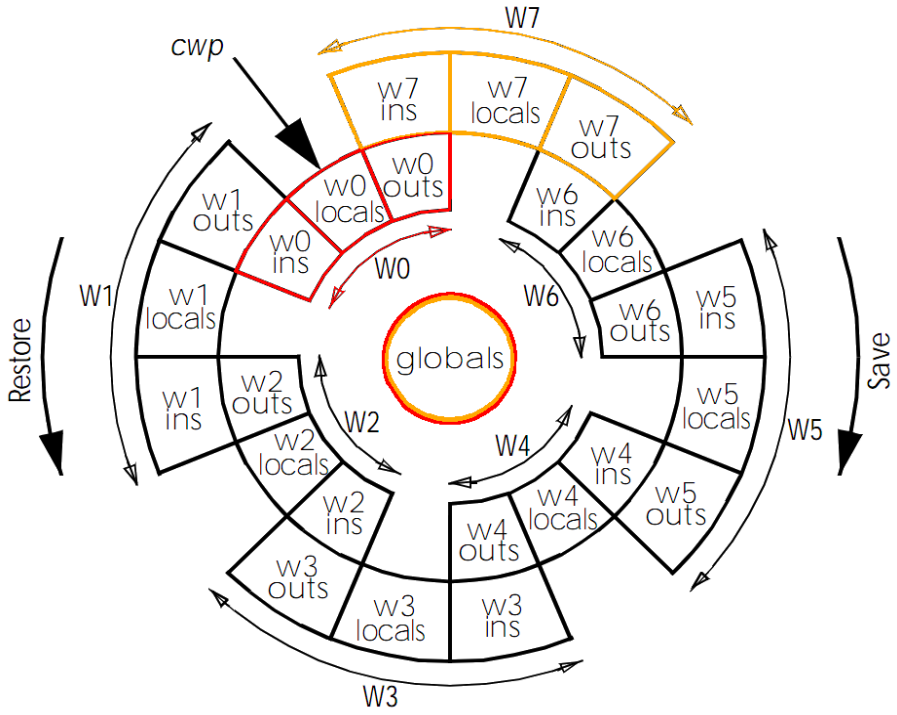


Figure 3.3 - Les fenêtres de registres du processeur LEON3

Les caractéristiques du processeur LEON3 peuvent être résumées comme suit :

- Ce processeur est organisé autour d'un système de 8 fenêtres. Chacune des fenêtres fournit un environnement de 32 registres de 32-bit distribués de la manière suivante:
    - 8 registres globaux (g0 à g7) : ces registres sont accessibles dans toutes les fenêtres à tout moment, ils sont donc partagés entre les fonctions.
    - 8 registres locaux (***local)*** l0 à l7 : ces registres sont spécifiques à chaque fenêtre, donc à chaque fonction.
    - 8 registres d'entrée (***input***) i0 à i7 : ces registres sont utilisés pour les paramètres passés par la fonction appelante à la fonction appelée et pour les retours de la fonction appelée vers la fonction appelante.
    - 8 registres de sortie (***output***) o0 à o7 : ces registres sont utilisés par les paramètres à passer vers une fonction appelée et pour les retours de la fonction appelée.
- Un appel de fonction ou une interruption provoquent un changement de la fenêtre de registre.
- Les registres d'entrée (*input*) de la fenêtre W_n deviennent les registres de sortie (*ouput*) de la fenêtre W_{n+1}. W_{n+1} reçoit un nouvel ensemble de registres locaux (*local*) et registres de sortie.

3.2.1.1 Passage d'une fenêtre à une autre

Comme mentionné précédemment, pour changer de fenêtre, deux appels assembleur sont permis :

- SAVE : cet appel est fait avant d'appeler une fonction. Dans ce cas :
  - Les registres globaux ne changent pas.
  - Les registres de sortie de la fenêtre courante deviennent (c'est à dire, sont renommés) les registres d'entrée de la nouvelle fenêtre.
  - Le processeur alloue des nouveaux registres locaux et de sortie (c'est à dire prend ceux de la fenêtre suivante).
- RESTORE : Cet appel est fait une fois que la fonction est terminée. Dans ce cas:
  - Les registres globaux ne changent pas.
  - Les registres d'entrée de la fenêtre courante deviennent (c'est à dire, sont renommés) les registres de sortie de la nouvelle fenêtre.
  - Le processeur "retrouve" les registres locaux et sortie de la fonction à laquelle on a retourné.

Dans ce qui suit est schématisé le passage d'une fenêtre à une autre lors d'un appel de fonction:

- La fonction est dans la fenêtre numéro 1, voir Figure 3.4 (a).
- La fonction va appeler une autre: elle met dans les registres de sortie les paramètres de celle-ci, fait un SAVE et appelle, voir Figure 3.4 (b).
- La nouvelle fonction a donc sa fenêtre, et les registres de sortie de la fonction d'avant sont devenus les registres entrée de celle-ci, voir Figure 3.4 (c).

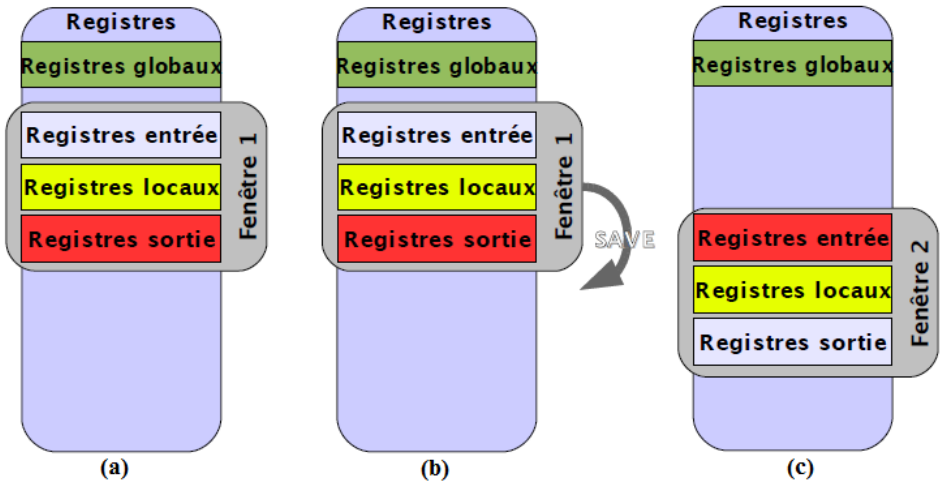


Figure 3.4 - Passage d'une fenêtre à une autre : L'appel de fonction

3.2.2 Les zones sensibles du processeur LEON3 aux SEUs

Comme décrit précédemment, pour émuler l'occurrence de SEUs pendant l'exécution de l'algorithme d'auto-convergence, nous avons utilisé l'approche CEU (Code Emulated Upsets). Les résultats de plusieurs campagnes d'injection de fautes issus d'expériences d'injection de fautes réalisées avec cette approche seront présentés à la fin de ce chapitre. Dans ce qui suit seront décrites les zones sensibles aux upsets du processeur LEON3.

Il est important de mentionner que parmi les 8 fenêtres de registres du processeur seule la fenêtre qui contient les variables du programme en cours d'exécution, c'est à dire la fenêtre liée à l'exécution du "main", sera considérée comme cible pour l'approche CEU, et parmi les registres globaux cinq seront utilisés par la procédure d'injection de fautes et donc ne seront pas cibles des upsets injectés. Les 8 fenêtres ensemble peuvent être ciblés par l'injection de fautes, mais ceci n'est pas fait puisque seule la fenêtre de référence est prise et comme les autres fenêtres ne sont utilisées lors de l'exécution de l'application, on sait que l'injection des fautes dans ces registres n'aura pas d'effets. Ces cinq registres globaux réservés par la procédure d'injection de fautes, contiendront, durant une campagne d'injection de fautes, l'adresse cible et le masque indiquant le bit à perturber. Les fenêtres des registres contiennent un total de 136 registres d'usage général (8 registres globaux + 128 registres de les fenêtres) dont, comme mentionné précédemment, cinq sont utilisés pour la méthode d'injection de fautes et ne sont donc pas considérés comme faisant partie de la zone sensible aux SEUs. Ainsi l'approche CEU permet injecter des fautes dans 27 registres de la fenêtre de registres considérée et si l'on considère toute la zone sensible, cette méthode permet injecter de fautes dans 131 registres. Dans la Figure 3.5 sont montrés les registres accessibles et non-accessibles en utilisant le jeux d'instructions du processeur LEON3.

Le processeur LEON3 contient deux compteurs de programme, PC (Program Counter) et nPC (next Program Counter) faisant partie aussi de la zone sensible. Ces deux registres sont localisés dans les registres locaux "l1" et "l2" de la fenêtre de registres de la routine d'interruption. Si l'un de ces deux compteurs est perturbé, le résultat peut aboutir à une perte de séquence ou un résultat erroné.

Les mémoires cache d'instructions et de données du LEON3 sont configurables (associativité, taille...) ayant chacune 1 Kb mappé directement. Dans le cas de l'approche CEU, les zones sensibles aux SEUs qui seront considérées dans les expériences d'injection de fautes sont les 2048 mots de 32 bits pour chacune des mémoires cache, 131 registres de la fenêtre de registres, le PC (Program Counter) et le nPC (next Program Counter) ce que fait un total de 2181 registres de 32 bits soit 69792 bits.

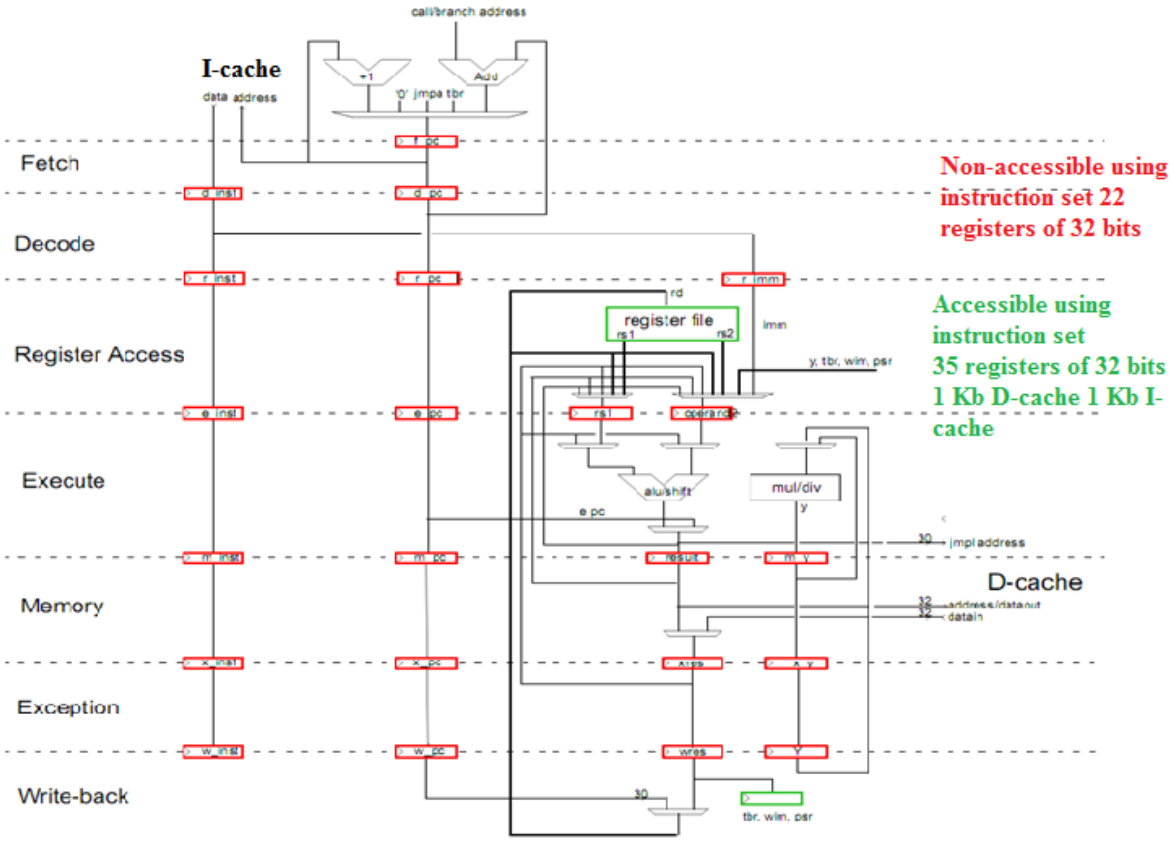


Figure 3.5 - Registres accessibles et non-accessibles du processeur LEON3

3.2.3 L'injection de fautes dans le processeur LEON3

Comme déjà mentionné, l'approche CEU a été choisie pour l'injection de fautes dans le processeur LEON3 lors de l'exécution d'une application donnée. Cette section a pour but donner des détails de point de vu pratique sur l'injection de fautes dans le processeur LEON3.

La Figure 3.6 donne le diagramme de la simulation des SEUs dans le processeur LEON3. Les étapes de la simulation sont résumées dans ce qui suit :

- Initialement une commande d'initialisation de la mémoire est envoyée au superviseur pour initialiser la mémoire partagé.
- Ensuite sont générés les paramètres d'injection de fautes: instant d'occurence du SEUs, l'adresse du registre cible et le bit à perturber (le masque des bits).
- Les paramètres d'injection de fautes sont alors stockés et l'application du LEON3 est lancée.
- L'interruption est donc générée en fonction du paramètre instant et la routine d'injection de fautes est exécutée, provoquant ainsi l'inversion du contenu du bit cible aléatoirement choisie.
- Retour vers le programme principal. L'application est exécutée par le LEON3 avec le SEU injecté.

- La fin d'application est détectée ou un timeout (perte de séquence, voir sous-section 3.3.2 pour plus de détails) est généré. Ensuite une commande de lecture de la mémoire est envoyée et les résultats sont donc comparés avec les résultats de référence.

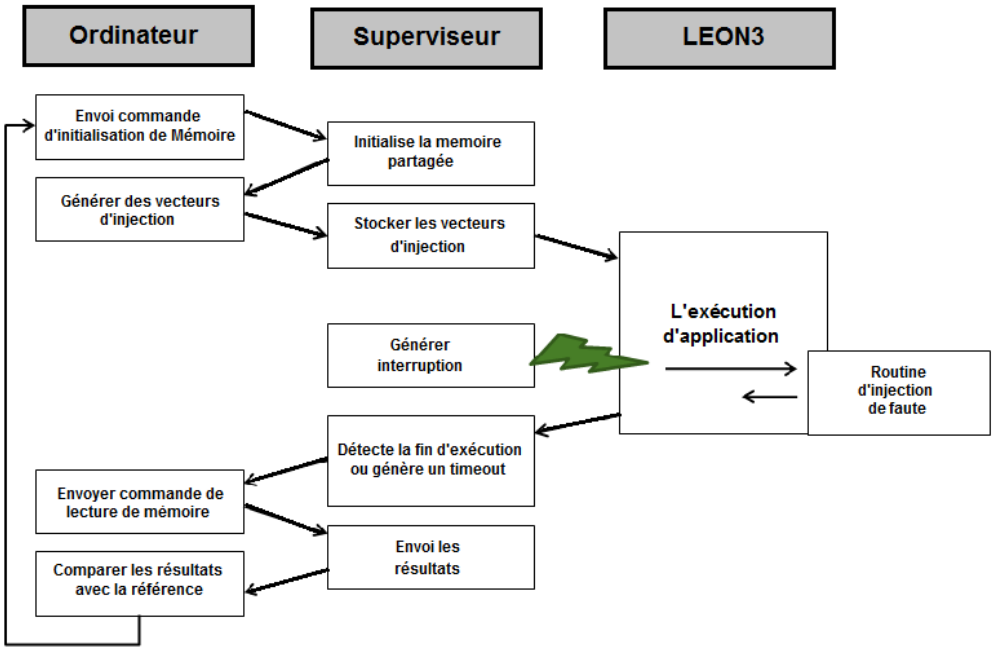


Figure 3.6 : Simulation des SEU's dans le processeur LEON3 lorsqu'il exécute une application donnée

Comme décrit précédemment, dans le cas de l'approche CEU, les zones sensibles aux SEUs du processeur LEON3 qui sont considérées dans les expériences d'injection de fautes sont les 131 registres de la fenêtre de registres, le PC (Program Counter), le nPC (next Program Counter) et 2048 mots de 32 bits pour chacune des mémoires cache. Dans ce qui suit sont données les routines d'interruption pour injecter un SEU dans les zones sensibles du processeur LEON3 accessibles pour l'approche CEU.

La routine d'interruption pour injecter un SEU dans les registres de la fenêtre de registres est décrite ci-dessous :

- Sauvegarder le status du processeur (%psr, Processor State Register).
- Lire le registre cible pour l'injection de la faute.
- Lire le registre de masque pour l'injection de la faute.
- Passer à la fenêtre W_{n+1} de programme.
- Sauter selon le registre cible.
- Une operation *BA* (Branch Always) est effectuée et provoque un transfer de contrôle retardé: le processeur récupère l'instruction suivante avant de sauter à l'adresse sélectionnée.

- Une opération logique *OU exclusif* (XOR) est effectuée entre le registre cible et une valeur de masque approprié.
- Revenir à la fenêtre W_{n-1} d'interruption.
- Restaurer le registre *%psr*.
- Sauter et connecter: retour au programme.
- Retour : passer la fenêtre W_{n+1} du programme.

Pour les deux compteurs de programme (PC et nPC) le processus d'interruption pour l'injection de SEU est donné dans ce qui suit :

- Sauvegarder le statut du processeur (*%psr*, Processor State Register).
- Lire le registre cible pour l'injection de SEU.
- Lire le registre de masque pour l'injection de SEU.
- Injecter une faute en tenant compte du masque: effectuer une operation logique OU exclusif (XOR) entre le registre (qui contient le PC ou nPC) et une valeur de masque appropriée.
- Passer à la fenêtre W_{n+1} de programme.
- Une operation *BA* (Branch Always) est effectuée et provoque un transfer de contrôle retardé: le processeur récupère l'instruction suivante avant de sauter à l'adresse sélectionnée.
- Revenir à la fenêtre W_{n-1} d'interruption.
- Restaurer le registre *%psr*.
- Sauter et connecter: retour au programme.
- Retour : passer la fenêtre W_{n+1} du programme.

Enfin, la routine d'interruption pour l'injection de SEU dans les caches d'instructions ou de données est décrite dans ce qui suit :

- Sauvegarder le statut du processeur (*%psr*, Processor State Register).
- Lire le registre cible pour l'injection de faute.
- Lire le registre de masque pour l'injection de faute.
- Lire l'adresse de mot de la cache choisi pour injecter le SEU. Cet instruction est utilisée avec les identifiants spécifiques d'espace d'adressage (*asi codes*) pour la cache d'instructions et pour la cache de données.
- Injecter une faute en tenant compte le masque: effectuée l'opération OU exclusive (XOR) entre l'adresse qui est applique le SEU et une valeur de masque approprié.
- Stocker l'adresse qui est appliqué le SEU.
- Passer à la fenêtre W_{n+1} de programme.
- Une opération *BA* (Branch Always) est effectuée et provoque un transfert de contrôle retardé: le processeur récupère l'instruction suivante avant de sauter à l'adresse sélectionnée.

- Revenir à la fenêtre W_{n-1} d'interruption.
- Restaurer le registre *%psr*.
- Sauter et connecter : retour au programme.
- Retour : passer la fenêtre W_{n+1} du programme.

Dans les ANNEXES C, D, E, F et G sont disponibles les codes pour l'injection de SEU dans les registres de la fenêtre de registres, dans les deux compteurs de programme (PC et nPC) et dans les mémoires cache d'instructions et de données.

3.3 L'environnement d'injection de fautes

Cette section a pour but de présenter l'environnement d'injection de fautes utilisé dans les campagnes de test effectuées sur les différentes implémentations de l'algorithme auto-convergeant étudié. Cet environnement est basé sur une plateforme de test générique appelée ASTERICS (*Advanced System for the TEst under Radiation on Integrated Circuits and Systems)* développée à TIMA pour la qualification de circuits intégrés digitaux face aux effets de radiations.

3.3.1 La plateforme de test: ASTERICS

Des tests dits « accélérés » sont nécessaires pour évaluer la sensibilité des composants face aux effets des radiations. Ces tests sont réalisés à l'aide d'accélérateurs de particules (ions lourds, protons, neutrons,…) qui permettent d'exposer à des flux importants de particules le circuit testé, pendant qu'il exécute une activité représentative. Ce tests étant de type *online* (le circuit cible est actif durant le test) nécessitent donc un environnement approprié pour que le circuit testé effectue une activité standard. Les tests en accélérateur de particules étant très coûteux, une alternative préliminaire consiste à émuler par injection de fautes les conséquences des particules. Pour effectuer de telles campagnes d'injection de fautes les cartes d'évaluation du commerce peuvent être utilisées, mais celles-ci n'offrent généralement pas la flexibilité de contrôle et d'observation des signaux et ressources sensibles du circuit. Une alternative est l'utilisation d'une plateforme de test fournissant au circuit sous test un environnement électronique approprié à son fonctionnement et qui permet à l'utilisateur la récupération des résultats obtenus lors de l'irradiation d'un circuit.

THESIC (Testbed for Harsh Environment Studies of Integrated Circuits) est un exemple représentatif d'une plateforme générique et flexible développée à TIMA pour la réalisation de tests de circuits intégrés en accélérateurs de particule [83]. Le but de THESIC était de permettre l'implémentation de l'environnement digital/analogique nécessaire pour l'opération du *DUT* (Device Under Test) via un FPGA dont sa configuration est obtenue par la compilation de cet environnement

décrit dans un langage de description du hardware tel que Verilog ou VHDL. Cette plateforme est composée principalement d'une carte mère bâtie autour d'un microcontrôleur Intel 80C51 et d'une carte fille mettant en application une architecture appropriée autour du processeur à tester [84].

La complexité croissante des circuits intégrés a obligé à améliorer les performances des différentes générations de testeurs développés à TIMA dans le but de faire face aux caractéristiques (fréquence, capacité de calcul...) des circuits fabriqués d'après les technologies nanométriques actuelles. La plateforme ASTERICS (Advanced System for the Test under Radiation of Integrated Circuits and Systems) dont une photo est montrée dans la Figure 3.7, est une version améliorée de la plateforme THESIC+, développée au laboratoire TIMA et présentée dans [85].

La plateforme ASTERICS est construite autour de deux FPGAs, le premier est appelé *Control FPGA,* et a pour tâche la gestion des communications entre l'ordinateur de l'utilisateur et les ressources disponibles sur la carte mère de l'ASTERICS, et inclut un *watchdog* programmable pour faire face, dans le cas où le DUT est un processeur, aux erreurs provoquant des *pertes de séquence*, détectant ainsi les fautes conduisent à des *timeouts*.

Figure 3.7 - Le testeur ASTERICS : carte mère

Les radiations peuvent provoquer des fautes destructives, SEL (Single Event Latchup) résultant du déclenchement de thyristors parasites et provoquant des courts-circuits capables de détruire le composant par effet thermique si le circuit n'est pas éteint dans le temps. La plateforme ASTERICS tient compte des fautes de type SEL par surveillance permanente de la consommation du courant du dispositif sous test afin de le protéger contre les fautes destructives, telles que les SELs, qui

peuvent se produire par exemple au cours des expériences en accélérateurs de particules, néanmoins elles n'ont pas été considérés dans cette thèse. Les transferts de données sont effectués via un réseau Ethernet assurant un transfert à haute vitesse. Les signaux d'entrée/sortie sont disponibles avec une palette de tensions d'alimentation allant de 1.2V à 3.3V et programmables par logiciel.

Le deuxième FPGA d'ASTERICS, appelé *Chipset FPGA*, contient le design développé par l'utilisateur qui peut être le *DUT* à tester, ou le design (contrôleur mémoire, superviseur du processus de l'injection des fautes, etc.) utilisé pour interfacer les ressources de la plateforme ASTERICS à une carte externe, appelée *carte fille*, construite autour du DUT. De cette manière sont réduits au minimum le temps, les efforts de développement et le coût de la plateforme matérielle nécessaire pour effectuer des essais sous radiation en accélérateur de particules ou des expériences d'injection de fautes sur un circuit particulier [86]. La Figure 3.8 donne un aperçu de l'architecture de la plateforme ASTERICS.

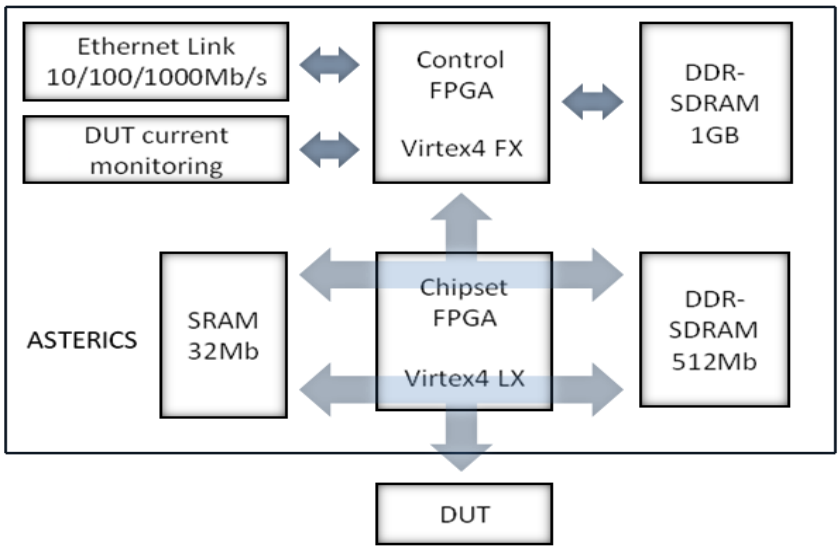


Figure 3.8 - L'architecture du testeur ASTERICS

3.3.2 L'installation du système: Setup

La plateforme ASTERICS a été utilisée comme plateforme matérielle pour l'injection de fautes dans le cadre de cette thèse. Comme dit précédemment la méthode appelée CEU (Code Emulated Upset), développée à TIMA pour simuler l'impact des fautes SEU dans des processeurs [78]. Ces fautes sont émulées par l'exécution d'un code approprié qui peut être activé à un instant choisi comme conséquence de l'assertion des signaux d'interruption asynchrones qui sont disponibles dans la plupart des circuits de type processeurs. Cette méthode est décrite avec détails dans le chapitre 3.1. Le processeur LEON3 (où le code VHDL est disponible sur [87]) a été implémenté sur le *Chipset* FPGA d'ASTERICS qu'exécutera l'algorithme d'auto-convergence présenté dans le chapitre 2.7. Un contrôleur mémoire commun, permettant l'accès à la SRAM du testeur à un ordinateur externe ainsi

qu'au LEON3, a été implémenté dans le *Chipset* FPGA. Le diagramme de l'installation du système Ordinateur-Testeur-LEON3 est donné dans la Figure 3.9.

Dans la Figure 3.9 est illustrée l'architecture de la plateforme expérimentale implémentée pour effectuer les campagnes d'injection de fautes de type SEU sur le LEON3. Le rôle du module *superviseur* inclut dans le *Chipset* FPGA est:

- Stocker les paramètres d'injection de fautes: instant de l'occurrence de SEUs, l'adresse du registre cible et le bit à perturber (le masque des bits). L'injection de fautes est effectuée par l'opération logique OU exclusif (XOR) entre le registre cible et une valeur de masque approprié (voir section 3.1).
- Lancer l'exécution de l'application du LEON3.
- Générer l'interruption en fonction du vecteur *instant*.
- Détecter la fin de l'application.
- Comparer les résultats obtenus avec les résultats de référence et compter les erreurs.

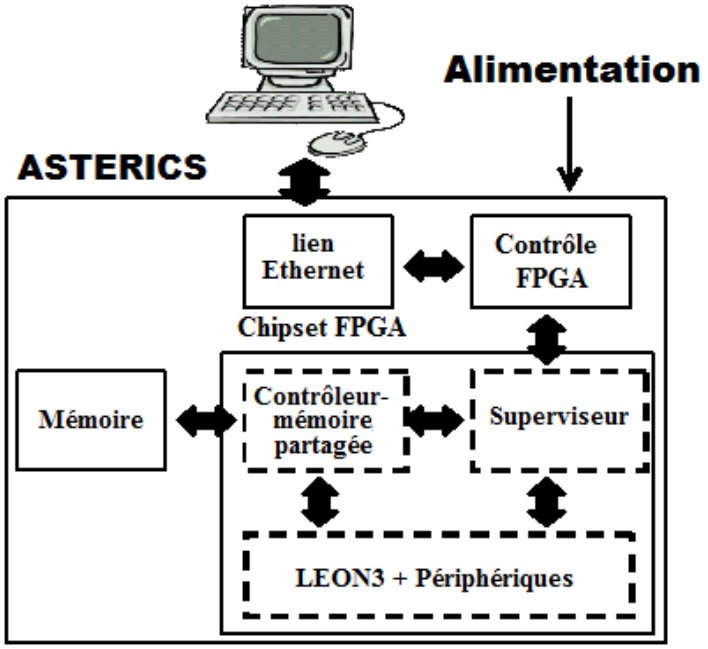


Figure 3.9 - Installation du système: L'ordinateur + Testeur ASTERICS + LEON3

Le superviseur a aussi comme but le traitement des timeouts (perte de séquence), erreurs qui peuvent être classées comme suit :

- Timeout de démarrage: détecte le moment où la séquence de démarrage ne se termine pas. Lorsque cela se produit, il faut relancer le programme.
- Timeout d'ASTERICS: détecte quand l'application en cours exécutée par le LEON3 ne se termine pas.

- Timeout de l'ordinateur: lorsque le superviseur ne fonctionne pas correctement ou l'ASTERICS ne répond plus dû à un bug ou à une perte de connexion.

Dans la Figure 3.10 est montré l'axe de temps du système d'injection de fautes avec les trois types de timeouts décrits ci-dessus. On peut constater à travers cette figure que la faute doit être injectée entre la fin du démarrage et la fin attendue de l'application. Pour ce faire il est nécessaire savoir quand l'exécution de l'application se termine.

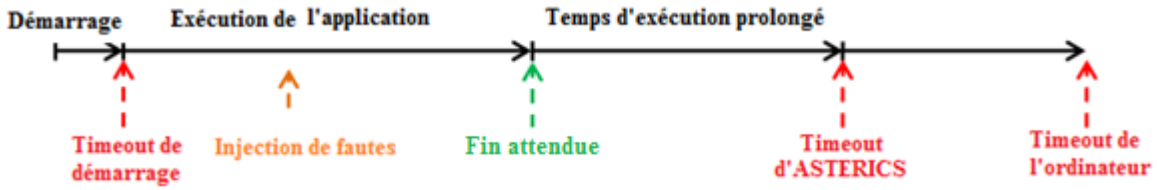


Figure 3.10 - Les trois différents types de timeouts de système

Un résumé de l'installation du système dans le cas du processeur LEON 3 est donné ci-dessous:

- Configuration matérielle: Ordinateur + Testeur ASTERICS + Source d'alimentation.
- Pas de carte fille DUT: *Chipset* FPGA utilisé comme DUT
- Mémoire ASTERICS: le code et données du LEON3
- Fonctions intégrées dans *Chipset* FPGA:
    - Contrôleur mémoire partagée: permettre l'accès par l'ordinateur et par le LEON3.
    - Superviseur: contrôle l'expérimentation.
    - LEON3 et ses périphériques.
- Programme exécuté par le LEON3: Un benchmark de l'algorithme auto-stabilisant.

3.3.3 Synthèse

Un résume de l'architecture du testeur ASTERICS est donné ci-dessous:

- Construite autour de deux FPGAs Virtex IV:
    - *Control* FPGA: XC4VFX60
    - *Chipset* FPGA: XC4VLX40
- Le testeur peut être contrôlé à distance via un réseau Ethernet 10/100/1000.
- L'utilisation du processeur PowerPC embarqué dans le FPGA pour contrôler le testeur.
- Jusqu'à 1 GB de mémoire DDR-SDRAM pour le *Control* FPGA.
- Mémoire Flash Compact utilisée pour stocker la configuration du FPGA et le programme du PowerPC.

- Jusqu'à 180 IOs disponible pour la connexion de DUT á le testeur via un connecteur à grande vitesse.
- Le DUT peut accéder à 32 Mb de mémoire SRAM et 512 Mb de DDR-SDRAM.
- La configuration du *Chipset* FPGA est gérée par le FPGA de contrôle.

Pour faciliter la communication avec la plateforme, une application API (*Application Programming Interface*) est disponible à l'utilisateur. Cette API est basée sur un ensemble de fonctions telles lecture/écriture de la mémoire, définition des limites de consommation de courant du DUT, réinitialisation du DUT, la configuration du *Chipset* FPGA, etc. Ces fonctions doivent seulement être utilisées par l'utilisateur dans leur application. Ces fonctions sont disponibles sous la forme d'une librairie dynamique DLL (*Dynamique Link Library*).

Pour conclure nous rappelons que deux caractéristiques importantes de la plateforme ASTERICS sont sa facilité d'adaptation à différents types de DUT et la possibilité de contrôler/observer de manière remote l'évolution d'une expérience de test (en accélérateur ou par injection de fautes) ceci via un lien ETHERNET et une connexion internet.

3.3.4 L'environnement de l'algorithme sous test dans le processeur

Comme décrit plus haut le processeur LEON3 a été implémenté dans le *Chipset* FPGA de la plateforme ASTERICS. Deux versions, l'une écrite en langage C et l'autre en assembleur, ont été utilisés pour obtenir les codes exécutables de l'algorithme d'auto-convergence étudié pour le LEON3.

Le programme exécuté par le processeur LEON3 a été compilé avec GCC (GNU Compiler Collection). En fait, on utilise un compilateur croisé pour SPARC. Les options, commandes, fichiers générés qui seront exécutées par *make* pour obtenir le fichier binaire de l'application sont mis dans un fichier appelé *Makefile*. Avec le fichier binaire il faut faire la configuration (de la limite d'exécution, choisir les zones cibles accessibles, des routines: pour la lecture et l'écriture des données dans la mémoire, pour avoir les fichiers de log ...) de logiciel de l'interface avec la plateforme ASTERICS pour avoir le code exécutable de l'algorithme sous test et ainsi lancer l'application pour une campagne d'injection de fautes.

La version originale de l'algorithme d'auto-convergence, sans l'implémentation de mécanismes de tolérance aux fautes, est présentée dans le chapitre 2.7. Sa version en langage assembler est donné dans l'ANNEXE H.

3.4 Campagnes d'injection des fautes

Dans cette section sont présentés d'abord les résultats d'une campagne d'injection de fautes pour l'algorithme de Dijkstra, ensuite ces résultats sont comparés aux résultats de plusieurs campagnes sur la version compilé en C justifiant le choix de l'algorithme d'auto-convergence, puis la version de l'algorithme travaillant directement en assembleur est présentée avec pour but l'évaluation de leur robustesse face aux fautes de type SEUs, et à la fin sont montrés les résultats des campagnes d'injection de fautes avec différentes stratégies (modifications logicielles et l'implémentation d'un TMR) avec le but d'améliorer les capacités de tolérance aux fautes de l'algorithme auto-convergeant face aux SEUs.

3.4.1 Résultats avec l'algorithme de Dijkstra

Les résultats des campagnes d'injection de fautes avec l'algorithme de Dijkstra ont comme but justifier le choix pour l'algorithme PCCAS d'auto-convergence comme plateforme d'experimentation. L'algorithme de Dijkstra, conçu par l'informaticien néerlandais Edsger Dijkstra en 1956 et publié en 1959 [88], est un algorithme classique, concerne le problème de plus court chemin dans un graphe orienté ou non orienté avec des arêtes de poids non-négatifs. L'algorithme de Dijkstra est l'un des algorithmes qui calcule le plus court chemin entre les sommets (ou nœuds) d'un graphe. Une fois qu'est choisi un sommet comme la racine de la recherche, cet algorithme calcule le coût minimum de ce sommet à tous les autres sommets du graphe. Même si L'algorithme de Dijkstra et le PCCAS donnent le même résultats, ils ne fonctionnent pas l'un et l'autre de la même manière. En particulier, l'algorithme de Dijkstra, à chaque étape, trouve un nœud particulier, dont il calcule la distance à la source; le PCCAS, comme dèja dit dans le chapitre 2, met à jour toutes les distances à partir des nouvelles données. Dans la section suivante on verra que le taux d'erreur obtenu avec le algorithme d'auto-convergence (PCCAS) en comparaison avec l'algorithme de Dijkstra sans aucune modification est assez inferieure, justifiant le choix pour le algorithme d'autoconvergence, PCCAS.

L'algorithme de Dijkstra commence avec une estimation initiale pour le coût minimum et ajuste successivement cette estimation. Cet algorithme considère qu'un sommet sera appelé *fermé* si le plus court chemin entre le sommet pris en tant que racine de recherche et un sommet cible est minimum. Sinon, il est dit *ouvert*. Dans ce qui suit est décrit le fonctionnent de l'algorithme.

Fonctionnement de l'algorithme:

Soit G (V,A) un graphe orienté et *s* un sommet de G:

1. Attribuer une valeur zéro à l'estimation de coût minimum du sommet *s* (la racine de recherche) et infini aux autres estimations.

2. Attribuer une valeur quelconque aux sommets précédents (le sommet précédent d'un sommet *t* est le sommet qui précède t dans le plus court chemin de *s* à *t*).
3. Tandis qu'il existe un sommet ouvert :
    - Soit *k* un sommet encore ouvert dont l'estimation est la plus faible de tous les sommets ouverts.
    - Fermer le sommet *k*.
    - Pour chaque sommet *j* qui est encore ouvert et qui est le successeur de *k* :
        - Ajouter l'estimation de sommet *k* avec le coût de l'arc qui relie *k* à *j*;
        - Si cette somme est meilleure que l'estimation précédente pour le sommet *j*, la remplacer et notez *k* comme sommet précédent de *j*,

Le code d'algorithme de Dijkstra est donné dans la Figure 3.11 et un exemple de l'algorithme peut être trouvé dans l'ANNEXE I. Le Tableau 3.1 donne les résultats d'une campagne d'injection de fautes réalisées avec la méthode CEU. Le temps d'exécution de cet algorithme est de 60,93 ms. Les conséquences des SEUs injectés lors de ces campagnes d'injection de fautes peuvent être classés comme suit [89] :

- Résultats erronés: si les résultats issus de l'exécution de l'application ne correspond pas aux résultats attendus.
- Timeouts: si la durée d'exécution de l'application est supérieure au timeout d'ASTERICS.
- Fautes silencieuses: La faute injectée n'a pas de conséquence sur les résultats de l'application. À titre d'exemple, les fautes silencieuses typiques sont celles qui affectent un registre ou données qui n'est pas utilisé par le processeur. Une autre possibilité est que l'algorithme fournit un résultat correct malgré la faute.

Tableau 3.1: Résultats d'injection de fautes sur l'algorithme de Dijkstra

# des Fautes Injectées	**Résultats Erronés**	**Timeouts**	**Fautes Silencieuses**
71389	15684 (21.97%)	7217 (10.11%)	48488 (67.92%)

Les résultats d'injection de fautes obtenus suite l'exécution de l'algorithme de Dijkstra montrent que l'algorithme a un taux d'erreurs de 32.08% (résultats erronés et timeouts).

```
T = N × N matrix
D = N ×1 matrix
D[0] = 0;
   for(i = 1; i < N; i++)
         D[i]  =  INFINIE;
   for(k = 1; k < N; k++){
         m = INFINIE
         for(i = 1;i < N;i++){
            if (D[i] == INFINIE){
               for( j = 0; j < N; j++){
                  if (m >= D[ j]+T[N * i + j]){
                        m = D[ j]+T[(N * i + j)];
                        min = i;
                   }
               }
            }
         }
         D[ min ] = m;
   }
```

Figure 3.11 - Le code C de l'algorithme de Dijkstra

3.4.2 Premiers résultats avec l'algorithme d'auto-convergence PCCAS

Plusieurs campagnes d'injection des fautes ont été effectuées pour l'explorer la robustesse intrinsèque de l'algorithme d'auto-convergence par rapport aux SEUs résultant des fautes injectées émulant le phénomène SEU. Il est important de noter que chaque fenêtre dispose de 32 registres et deux parmi ces registres (%i6 et %o6, respectivement *frame pointer* et *stack pointer*) ne sont pas considérés comme cibles pour l'injection de fautes car ils peuvent affecter la méthode CEU. Le processeur LEON3 a une architecture un peu différente des processeurs typiques surtout dans la zone concernant la fenêtre de registres, donc lorsque un signal d'interruption est activé la fenêtre de registres du LEON3 est changée, évitant ainsi l'utilisation d'une pile pour enregistrer le contexte d'être restaurée après que le programme d'interruption est exécuté. Donc, ces premières campagnes ont ciblé uniquement les 25 registres de la fenêtre de registres contenant les variables du programme. Les résultats de ces premières campagnes de test ont donné lieu à une première publication présentée dans la conférence LATW'11 (Latin American Test Workshop) [89].

Plusieurs *runs*, au cours desquels des SEUs ont été injectés de façon aléatoire dans le temps et le lieu, ont été réalisés. Il est important de noter que dans chacune de ces exécutions un seul SEU par exécution a été injecté dans la zone sensible considérée du LEON3. En d'autres termes, un seul bit du registre choisi au hasard de la fenêtre de registres du LEON3 a été perturbé à un cycle d'horloge choisi au hasard dans le temps d'exécution du programme étudié. Il est important de noter que le taux

d'injection de fautes dans le cas de l'algorithme d'auto-convergence a été d'environ 1 SEU toutes les 2 secondes.

Comme les fautes injectées peuvent perturber la durée d'exécution de l'algorithme testé et l'algorithme d'auto-convergence est supposé faire face aux fautes en ré-exécutant les boucles jusqu'à la stabilisation, un paramètre important est la limite d'exécution (*running limit*) considérée pour identifier un timeout en cas de dépassement. Dans ces campagnes, les fautes SEU ont été injectées à des instants choisis dans la durée nominale du programme exécuté. Les résultats préliminaires explorent différentes limites de fonctionnement ceci pour obtenir une évaluation de l'impact des fautes injectées sur la durée d'exécution. La limite d'exécution varie d'une campagne à une autre, étant respectivement 1.5, 5, 8 et 16 fois la durée nominale du programme, qui est égale à 336 ms.

Dans le Tableau 3.2 sont montrés les résultats des campagnes d'injection des fautes en fonction du temps d'exécution de l'algorithme. Les conséquences des SEU injectés lors de ces campagnes d'injection de fautes sont les mêmes que celles présentés dans la section 3.4.1.

Comme le montrent les deux premières lignes du Tableau 3.2, les essais 1 et 2 ont été réalisées avec une limite d'exécution de 1.5 de la durée nominale. Les résultats obtenus dans ces cas, montrent que très peu des résultats erronés, environ 0.15%, ont été détectés. A l'opposé, environ 25% de fautes injectées ont provoqué des timeouts. Un si grand nombre des timeouts n'est pas objectif et certainement cache des résultats erronés. En effet, dans la présence de fautes, l'algorithme d'auto-convergence testé dans ces campagnes prend très souvent plus que 1.5 de temps d'exécution nominal pour converger vers un fonctionnement correct.

Dans les essais 3, 4 et 5, la durée d'exécution était significativement plus élevée, respectivement 5, 8 et 16 fois la durée de référence. Ces essais ont clairement mis en évidence une augmentation significative des résultats erronés, prouvant ainsi que les timeouts masquent des telles erreurs dans le cas où la limite d'exécution n'est pas appropriée. Dans ces premières campagnes, les résultats erronés et les timeouts ont été d'environ 6% et 11% respectivement. Dans le Tableau 3.2 on peut remarquer que les taux d'erreurs et des timeouts deviennent stables si la durée d'exécution est supérieure ou égale à 5 fois la durée nominale du programme. Autrement dit, si la durée d'exécution de l'application est prolongée davantage, ceci ne va pas avoir des effets remarquables sur les résultats.

Les résultats obtenus suite à ces campagnes préliminaires d'injection de fautes confirment la capacité de l'algorithme d'auto-convergence à fournir des résultats corrects même en présence de fautes affectant les données et les registres utilisées par l'application. En plus, il est montré que la durée d'exécution peut cacher des erreurs qui ont été considérées comme timeouts dans des campagnes ayant une "limite d'exécution" plus petite.

Tableau 3.2: Résultats de l'injection des fautes sur l'algorithme d'auto-convergence

Test	**# des Fautes Injectées**	**Résultats Erronés**	**Timeouts**	**Fautes Silencieuses**	**Limite d'exécution**
1	130577	204 (0.15%)	32143 (24.61%)	98230 (75.24%)	1.5
2	199550	324 (0.16%)	49478 (24.79%)	149748 (75.05%)	1.5
3	15068	1709 (11.34%)	992 (6.58%)	12367 (82.08%)	5
4	14264	1614 (11.31%)	900 (6.31%)	11750 (82.38%)	8
5	8007	887 (11.08%)	508 (6.34%)	6612 (82.58%)	16

Dans notre cas, pour $N = 16$, les données d'entrée (matrice T) représentent 256 entiers et 21 variables représentent 16 entiers par la sortie D et 5 entiers par les variables du programme, tous ces paramètres sont codés en 32 bits. Dans le Tableau 3.3 on peut voir les variables de l'algorithme d'auto-convergence ainsi que les types d'erreurs observées sur chacune de ces variables lors des campagnes d'injection de fautes et si elles peuvent être récupérable ou non [90].

Tableau 3.3: Variables sensibles de l'algorithme d'auto-convergence

Variables	**Types d'erreurs observées**	**Récupérable**
T	Timeouts et résultats erronés	non
D	Timeouts et résultats erronés	non
i	timeouts	oui
j	timeouts	oui
m	Timeouts et résultats erronés	oui
b	Timeouts et résultats erronés	oui
c	Timeouts et résultats erronés	oui

Les erreurs étant injectées aléatoirement dans le temps et dans la localisation, la plupart de fautes injectées vont toucher les matrices T et D, ce qui explique un taux de résultats erronés plus élevé que celui attendu qui est égale à 11%.

Les erreurs affectant les données, c'est à dire les éléments des matrices *D* et *T*, ne peuvent pas être corrigées. Si ces données sont modifiées, le résultat est calculé conforme à la modification de la donnée et donc n'est pas supposé être récupérable par l'algorithme d'auto-convergence. En d'autre termes, l'algorithme converge mais pas au bon résultat. A l'opposé, les fautes de type *timeout*, peuvent toutes être détectées via un "watchdog timer".

De façon générale, on peut dire que les variables qui peuvent être « durcies » sont celles qui ne dépendent pas directement de l'entrée et de la sortie de l'application. Ceci pour éviter que le système commence les calculs avec des données erronées, ou qu'il enregistre des sorties incorrectes dans la mémoire cache des données du processeur.

Ces résultats offrent une bonne évaluation de la robustesse de l'algorithme d'auto-convergence étudié en ce qui concerne les erreurs de type SEU induites par le rayonnement naturel. Environ 11% des fautes injectées, ont provoqué des résultats erronés de l'algorithme d'auto-convergence, alors que les timeouts représentent 6% des fautes injectées. Ainsi, un pourcentage significatif des SEUs est toléré par la version originale (sans implémentation de mécanismes de tolérance aux fautes) de l'algorithme d'auto-convergence étudié. Ces résultats obtenus peuvent être comparés aux résultats de l'injection des fautes sur l'algorithme de Dijkstra présentés dans la sous-section 3.4.1. Cet algorithme ainsi comme l'algorithme d'auto-convergence (appelé PCCAS) calcule le plus court chemin entre deux nœuds, on peut voir que l'algorithme de Dijkstra a un taux d'erreurs de 32.08% et le PCCAS a un taux d'erreurs de 17.92%, c'est à dire que l'algorithme d'auto-convergence apporte un gain de 14.16% justifiant le choix de cet algorithme dans ces travaux. Ces résultats donnent une bonne perspective pour atteindre la robustesse requise par les applications critiques en incluant dans l'algorithme d'auto-convergence des techniques logicielles de tolérance aux fautes.

3.4.3 Résultats obtenus avec l'algorithme d'auto-convergence implémenté au niveau assembleur

Le but ici est d'étudier les moyens d'améliorer les capacités de tolérance aux fautes de l'algorithme d'auto-convergence par la conversion et l'amélioration du logiciel au niveau du langage assembleur. En effet, en travaillant directement au niveau du langage assembleur il est possible d'optimiser manuellement l'attribution des registres aux variables du programme. La connaissance des registres ciblés au cours des campagnes d'injection de fautes permet d'obtenir une «pathologie» des erreurs qui échappent aux capacités de tolérance aux fautes de l'algorithme d'auto-convergence.

Des campagnes d'injection des fautes avec l'approche CEU ont été effectuées pour évaluer la sensibilité aux SEUs des variables du programme et pour identifier les cibles les plus sensibles aux SEUs. Ces expériences ont prouvé que les SEUs injectés peuvent amener à des résultats erronés ou à des timeouts. Les SEUs qui affectent les variables du programme (*i*, *j*, *m*, *b* et *c*) sont en principe

récupérables, tandis que ceux affectant les données, telles que les éléments des matrices *T* et *D*, ne peuvent pas être corrigés par l'algorithme auto-convergeant. Les éléments des matrices *T* et *D* ne seront donc pas considérés comme cibles pour les expériences d'injection de fautes.

L'algorithme étudié est présenté à nouveau dans la Figure 3.12 ci-dessous, afin d'identifier les variables et données auxquelles on fera allusion dans les analyses qui seront fait par la suite :

```
b = c = 1
T = N × N matrix
D = N ×1 matrix
while(b || c){
   c = b;
   b = 0;
  D[0] = 0;
  for(i = 1; i < N; i++){
     m = INFINIE;
    for( j = 0; j < N; j++){
      if (m >= D[j]+T[N * i + j])
          m = D[j]+T[N * i + j];
    }
    if (D[i] != m)
      b = 1;
   D[i] = m;
 }
```

Figure 3.12 - Le code C de l'algorithme d'auto-convergence PCCAS

Les conséquences des SEUs injectés sont classifiées de la même manière que dans la sous-section 3.4.1, après les analysis des premières résultats pour cet algorithme, comme indiqué précédemment, le taux d'erreurs se stabilisent avec une durée d'exécution égale à 5 fois la fin attendue (la durée d'exécution nominale de cet algorithme au niveau du langage assembleur est de 47,268 ms). En outre, il a été observé aussi que l'algorithme peut fournir des résultats corrects après le temps limite prévu en cours d'exécution, donc une quatrième classification a été considérée et appelé de *convergence*. La *convergence* représente les fautes qui ont été tolérées par l'algorithme étudié.

Le Tableau 3.4 présente les résultats d'une première campagne d'injection de fautes effectuée sur l'algorithme d'auto-convergence original écrit au niveau de l'assemblage sans aucune implémentation supplémentaire de tolérance aux fautes. Dans cette campagne, les fautes ont été injectées dans les 25x32 bits des registres actifs de la fenêtre des registres. Il est important de rappeler que chaque fenêtre du processeur LEON3 dispose de 32 registres, mais cinq parmi ces registres ne

peuvent pas être ciblés, car ils sont utilisés à des fins d'injection de fautes. La limite d'exécution a été choisie égale à 5 fois le temps d'exécution nominale, ceci pour les mêmes raisons que celles évoquées auparavant. On peut constater que l'algorithme d'auto-convergence détecte et corrige 12.83% de fautes injectées.

Tableau 3.4: Résultats de l'injection de fautes

# des Fautes Injectées	Résultats Erronés	Timeouts	Fautes Silencieuses	Convergences
17608	833 (4.73%)	2340 (13.29%)	12176 (69.15%)	2259 (12.83%)

La Figure 3.13 montre la distribution de SEUs dans les registres actifs de la fenêtre de registres.

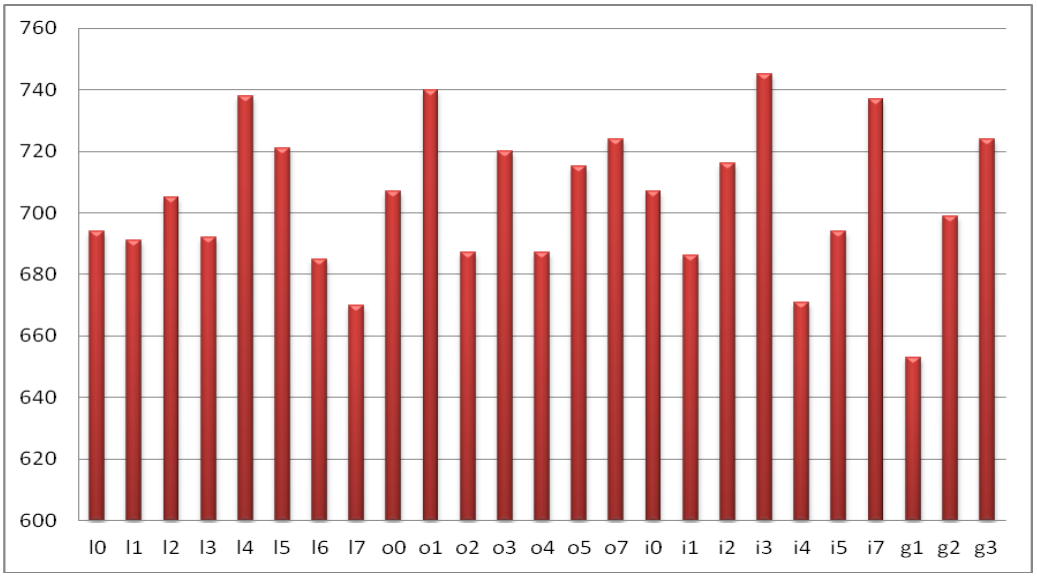


Figure 3.13 - Distribution des SEUs dans les registres actifs

Les Figures 3.14 et 3.15 montrent la distribution d'erreurs et de timeouts dans les registres actifs de la fenêtre de registres résultant de l'injection de fautes sur l'algorithme d'auto-convergence.

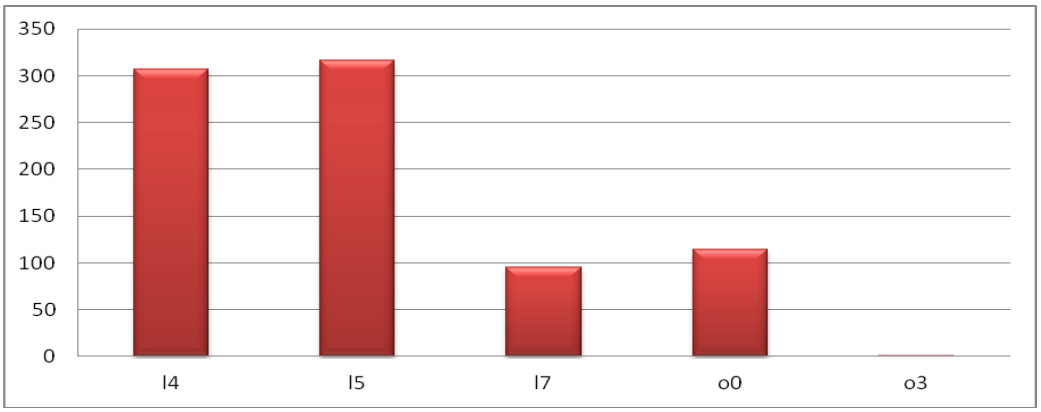


Figure 3.14 - Distribution des SEUs provoquant des résultats erronés de l'algorithme d'auto-convergence: registres actifs vs. fautes injectées

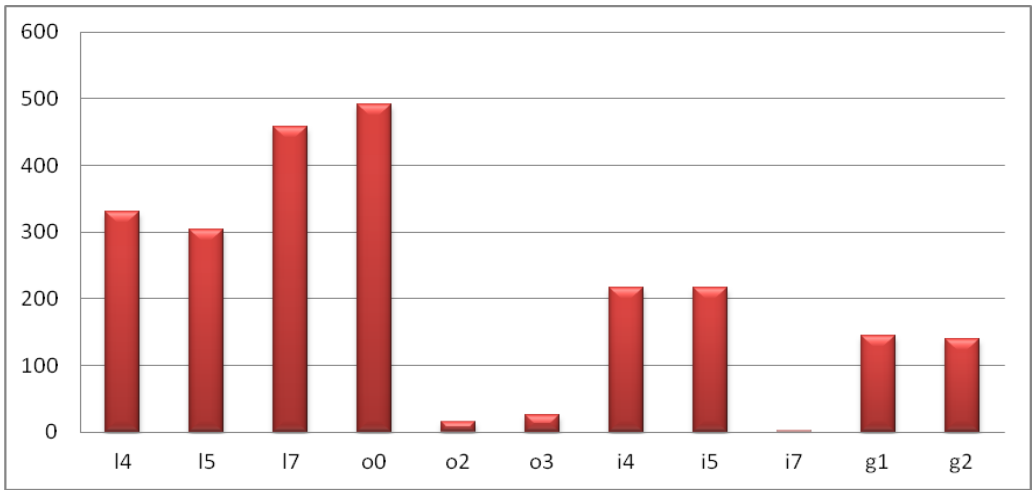


Figure 3.15 - Distribution des SEUs provoquant timeouts de l'algorithme d'auto-convergence: registres actifs vs. fautes injectées

Les résultats de l'analyse montrent que les variables plus susceptibles de provoquer des résultats erronés et des timeouts sont les variables *b* (%l4), *c* (%l5) et *j* (%l7). Les variables *b* et *c* sont utilisées pour détecter la convergence de l'algorithme, mais si *b* et *c* sont mises à zéro indûment, elles conduisent à des résultats erronés.

Les résultats erronés causés par *i* ou *j* s'expliquent si dans une itération la variable *i* et/ou *j* est fixée à une valeur différente de la valeur prédite dans la boucle. Les variables *c*, *j* et *i* provoquent des timeouts dans les cas suivants : la variable *c* est fixée à une valeur différente de zéro, les variables *i* ou *j* sont fixées à une valeur inférieure à *N* durant une itération.

Comme les erreurs peuvent se produire lorsque les SEUs corrompent les adresses de base des matrices *T* et *D*, une première modification a été réalisée: les adresses de base des matrices d'entrée et

de sortie sont réinitialisées avant chacune des exécutions des instructions *load* et *store* concernant ses éléments. Un échantillon de ce processus est donné au-dessous :

```
set @addr,%r1                set @addr,%r1
set @addr,%r2                set @addr,%r2
ld[%r1+%r3],%r4              st %r3, [%r2]
```

Compte tenu que l'algorithme d'auto-convergence ne peut pas tolérer certaines fautes perturbant le contenu des compteurs des boucles *i* et *j*, une deuxième modification a été effectuée sur le code assembleur. Cette modification s'agit d'une implémentation TMR (triplication et vote) des registres contenant les variables *i* and *j*. Une dernière modification de l'algorithme d'auto-convergence étudié a été d'initialiser les variables *b* et *c* avec une valeur autre que «1», cela pour éviter qu'ils deviennent «0» et provoquent des délais d'attente comme conséquence d'un SEU.

Une campagne d'injection de fautes ciblant les registres actifs de la fenêtre de registres a été réalisée pour obtenir une évaluation de l'impact des modifications effectuées au niveau du code assembleur sur les capacités de tolérance aux fautes de l'algorithme d'auto-convergence. Le Tableau 3.5 montre que le taux de résultats erronés et les timeouts diminuent de façon très significative: de 4.73% à 0.01% et de 13.29% à 4.61% respectivement. La durée d'exécution nominale de cet algorithme au niveau du langage assembleur avec les modifications présentées est de 73,175 ms.

Tableau 3.5: Résultats de l'injection de fautes sur le code assembleur modifié

# des Fautes Injectées	Résultats Erronés	Timeouts	Fautes Silencieuses	Convergences
43677	5 (0.01%)	2014 (4.61%)	36464 (83.49%)	5194 (11.89%)

Une dernière campagne d'injection de fautes ciblant toute la zone sensible accessible a été réalisée pour obtenir une évaluation de l'efficacité des modifications faites au niveau du langage d'assemblage dans le logiciel exécuté. Les résultats de cette campagne sont résumés dans le Tableau 3.6. La Figure 3.16 montre la distribution de fautes pour l'ensemble de ressources (RF, DC, IC, PC et nPC) sensibles du processeur LEON3 accessible par la méthode CEU. L'analyse des résultats montre que sur un total de 21343 fautes injectées, 402 conduisent à des résultats erronés : 1 provient de la fenêtre de registres (RF *register file*) 44 proviennent de la cache d'instructions (IC) et 357 de la cache de données (DC).

Parmi les fautes injectées, 534 conduisent à des timeouts. 9 proviennent de la fenêtre de registres (RF), 334 de la cache de données (DC), 179 de la cache d'instructions, 3 du PC (Program Counter) et 9 du nPC (next Program Counter).

Tableau 3.6: Résultats de l'injection de fautes, dans le code assembleur modifié, sur toutes les ressources accessibles du processeur LEON3

# des Fautes Injectées	Résultats Erronés	Timeouts	Fautes Silencieuses	Convergences
21343	402 (1.9%)	534 (2.5%)	19454 (91.14%)	953 (4.46%)

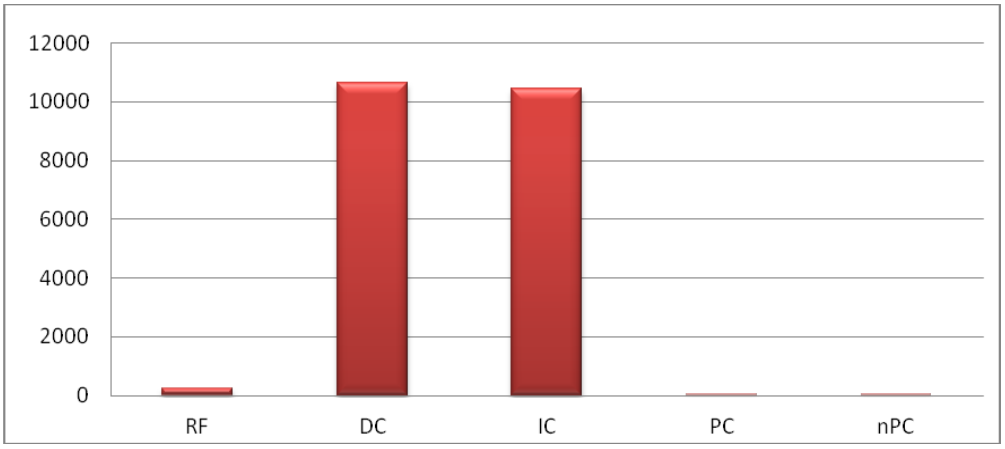


Figure 3.16 - Distribution des SEUs pour l'ensemble de ressources accessibles du processeur LEON3

Les Figures 3.17 et 3.18 montrent la distribution d'erreurs et de timeouts sur toute la zone sensible du processeur issue de l'injection de fautes sur l'algorithme d'auto-convergence avec le code assembleur modifié.

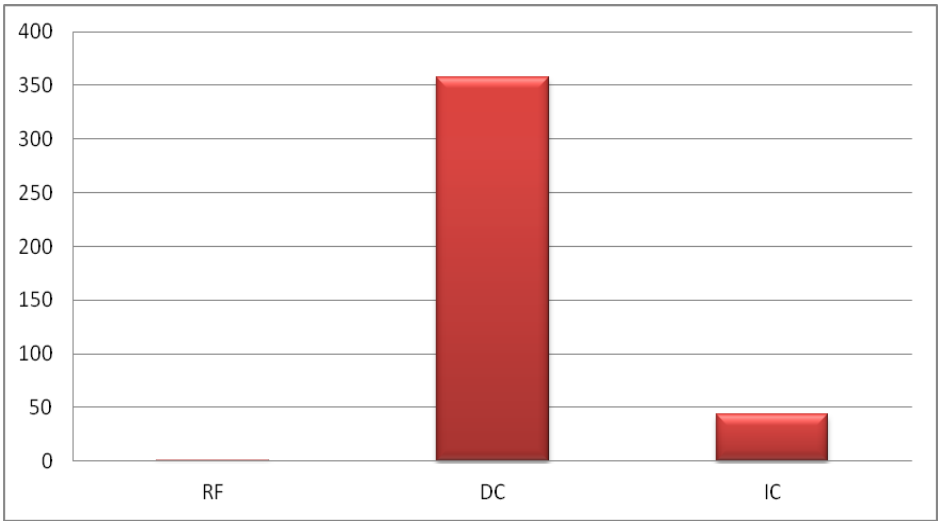


Figure 3.17 - Distribution des SEUs provoquant des résultats erronés de l'algorithme d'auto-convergence modifié au niveau de l'assemblage: ensemble de ressources vs. fautes injectées

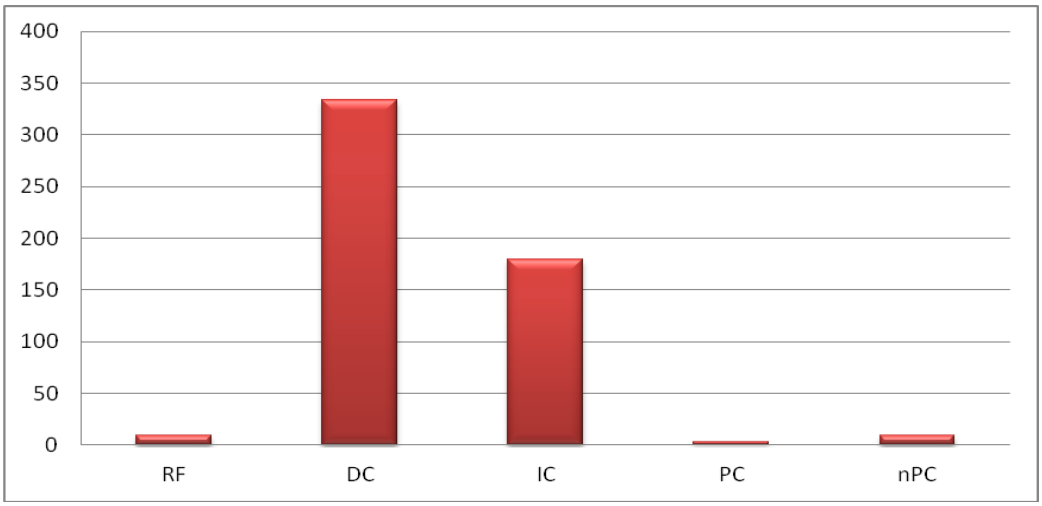


Figure 3.18 - Distribution des SEUs provoquant timeouts de l'algorithme d'auto-convergence modifié au niveau de l'assemblage: l'ensemble de ressources vs. fautes injectées

Les résultats obtenus avec cette technique de tolérance aux fautes ont donnés lieu à une troisième publication présentée à la conférance LASCAS (Latin American Symposium on Circuits and Systems) [91].

3.4.4 Fautes échappant à la technique de tolérance aux fautes

Pour effectuer l'anlyse des fautes injectées qui échappent à la technique de tolérance aux fautes on a pris les résultats de campagnes d'injections de fautes ciblant les registres de la fenêtre de registres, car les contenus des registres sont connus d'avance. Le Tableau 3.7 donne la distribution des SEUs dans les registres actifs de la fenêtre de registres ainsi comme la distribution des erreurs (résultats erronés) et des timeouts dans ces registres, issus de l'injection de fautes sur l'algorithme d'auto-convergence avec le code assembleur modifié.

Une explication pour certaines des fautes provoquant des résultats erronés a été trouvée. Ce cas concerne toutes les fautes qui provoquent la fin de la boucle avant le temps nécessaire pour terminer l'exécution de l'application, dans ce contexte deux situations se distinguent :

- Corruption de l'instruction *OR*: Le registre où l'erreur est identifiée, le registre ***%o3*** il y a dans une instruction *OR* avec les variables *b* e *c*, donc pour provoquer une erreur dans les variables *b* e *c* elles doivent toutes les deux être mis à «0». La probabilité que cette situation devienne vraie est faible car: ces variables dans la version durcie de l'algorithme d'auto-convergence sont initialisées avec un entier de 32-bits au lieu d'être initialisées avec une valeur ayant un seul bit «1» comme auparavant, et la variable *c* prend toujours la valeur que *b* avait dans l'itération précédente. Alors les 4 erreurs perturbant le registre *%o3* son dues à la corruption par un SEU de l'instruction

assembleur *OR*. Une deuxième situation concerne les fautes qui affectent l'ancienne valeur de *D*. En effet, l'ancienne valeur de *D* est modifiée après le calcul de la distance *m* (m=min{D[j]+T[i,j]}) au cours de l'itération courante, passant à avoir exactement la même valeur que celle de la nouvelle *D*, donc avec la valeur ancienne et la valeur courante égales on sort de la boucle avant le temps de finir l'exécution de l'algorithme.

Table 3.7: Distribution des SEUs dans les registres actifs: SEUs provoquant des résultats erronés et des timeouts de l'algorithme d'auto-convergence modifié au niveau de l'assemblage

Registres	# des Fautes Injectées	Résultats Erronés	Timeouts
l0	1784	0	0
l1	1700	0	0
l2	1769	0	0
l3	1736	0	0
l4	1809	0	54
l5	1797	0	36
l6	1738	0	0
l7	1777	0	25
o0	1767	0	27
o1	1685	0	0
o2	1747	1	47
o3	1757	4	51
o4	1733	0	0
o5	1739	0	0
o7	1624	0	0
i0	1795	0	0
i1	1670	0	0
i2	1706	0	0
i3	1761	0	0
i4	1794	0	552
i5	1752	0	557
i7	1748	0	0
g1	1811	0	336
g2	1778	0	329
g3	1700	0	0

Dans le cas où des basculements des bits échappent à la technique de tolérance aux fautes et ont pour conséquence des délais d'attente (timeouts), l'algorithme entre dans une boucle inifinie. Cette situation peut apparaître en un des cas suivants:

- Les variables *b* et *c* dans le temps d'une itération ne passent jamais à faux, c'est à dire l'application reste toujours dans la boucle *while*, la variable *b* est fixée à zéro et par

conséquence la variable *c*, qui prend la valeur qu'avait *b* à l'itérerions précédente, ne conduit pas à la fin du programme.

- Les variables *i* et *j* (les compteurs de boucle) sont fixées à une valeur inférieure à *N* dans le temps d'une itération, ainsi l'algorithme ne peut pas sortir de la boucle.

Un autre cas de perte de séquencement est dû aux instructions de démarrage de l'application. Cette type de perte de séquence est classée comme *timeout de démarrage*, c'est à dire quand la séquence de démarrage ne se termine pas, dans ce cas il faut relancer l'application.

Dans le Tableau 3.8 est donnée la classification des fautes qu'échappent aux techniques de tolérance aux fautes implémentées.

Tableau 3.8 : Classification des fautes qui échappent à la technique de tolérance aux fautes

La cause provocant des erreurs non détectées	**Type de fautes non détectée**	
	Résultats Erronés	**Timeouts**
Corruption de l'instruction *OR*	80%	-
L'ancienne valeur de *D* est corrompue	20%	-
Les variables *b* et *c* ne passent jamais à faux	-	8.34%
Les variables *i* et *j* sont fixée à une valeur inférieure à *N*	-	58.64%
Instructions de démarrage	-	33.02%

3.4.5 Résultats obtenus pour une implémentation multicore

Des améliorations significatives de la tolérance au fautes de type SEU d'un algorithme d'auto-convergence sont obtenues si différentes stratégies (modifications du code C de l'algorithme d'auto-convergence, sélections des registres et TMR) sont combinées. Une première modification a été effectuée sur le code C de l'algorithme d'auto-convergence. Un opérateur *modulo* "%" supplémentaire de 256 et 16 est ajouté à l'intérieur de chaque appel des entrées et sorties, respectivement, pour garantir que l'adressage sera toujours dans la zone mémoire correcte, ce qu'assure une probabilité presque nulle d'avoir des entrées erronées ou d'écrire dans des zones mémoires inattendues. Un exemple de l'utilisation de cet opérateur est donné par le remplacement de la douzième ligne de l'algorithme (voir Figure 3.12) par l'instruction suivante :

$$m = D[j\%16] + T[((N * i) + j)\%256]$$

Dans la Figure 3.19 est donné le code C de l'algorithme d'auto-convergence incluant la modification mentionnée ci-dessus. Il est important de dire que les entrées (matrice *T*) sont 256 variables et les sorties (matrice *D*) sont 16 variables, et alors la zone d'adressage des ces deux matrices est de l'ordre de leurs dimensions.

Une autre modification consiste à attribuer un registre local dans la fenêtre des registres à chaque variable du programme. Les registres locaux choisis sont les registres *l0* à *l4* de la fenêtre des registres, avec cette modification les ressources utilisées par la fenêtre sont réduites au minimum. Pour effectuer cette modification l'instruction suivante a été utilisée :

$$register\ unsigned \text{int}\ variable\ asm("register\ name")$$

```
b = c = 1
T = N × N matrix
D = N ×1 matrix
while(b || c){
   c = b;
   b = 0;
  D[0] = 0;
  for(i = 1; i < N; i++){
     m = INFINIE;
    for( j = 0; j < N; j++){
      if (m >= D[j%16]+T[((N * i) + j)%256])
         m = D[j%16]+T[((N * i) + j)%256];
    }
    if (D[i%16] != m)
      b = 1;
   D[i%16] = m;
 }
```

Figure 3.19 - Le code C de l'algorithme d'auto-convergence avec l'opérateur modulo "%"

Enfin, la même modification présentée dans la section précédente est considérée ici pour éviter les timeouts produits par les SEUs qui affectent les variables *b* et *c*. On initialise chacune de ces variables avec une valeur autre que «1». Pour l'algorithme d'auto-convergence étudié la valeur choisie pour chacune de ces variables est un entier de 32-bits aléatoire. Pour ces variables on a choisie leur valeur égale à «0xFA7F». De cette façon ces variables ne peuvent affecter la boucle parce qu'un SEU ne peut faire leurs contenus égaux à zéro en garantissant donc que la faute ne provoque pas une sortie erronée de la boucle.

Une campagne d'injection de fautes a été effectuée pour mettre en évidence l'efficacité de l'algorithme d'auto-convergence tandis que le LEON3 exécute le code modifié. La limite du temps d'exécution choisi est égale à 5 fois le temps d'exécution nominale, car comme le montre le Tableau 3.2 (sous-section 3.4.2), avec cette limite les taux des résultats erronés et des timeouts se stabilisent. Dans le Tableau 3.9 sont présentés les résultats de cette campagne d'injection de fautes réalisée sur l'algorithme modifié. Les résultats obtenus ont mis en évidence une diminution significative des résultats erronés et des timeouts. Les erreurs passent de 11.34% à 4.45% alors que les timeouts passent de 6.58% à 2.6%. L'algorithme d'auto-convergence détecte et corrige 37.15% des fautes injectées, ce qui montre que les modifications effectuées dans le code source de l'algorithme apportant un gain total de 10.87% (ce résultat est la différence entre la somme des taux des résultats erronés et des timeouts des campagnes utilisant l'algorithme sans durcissement et l'algorithme avec les modifications présentées).

Tableau 3.9 : Résultats de l'injection des fautes sur le code C modifié

# des Fautes Injectées	Résultats Erronés	Timeouts	Fautes Silencieuses	Convergences
8000	356 (4.45%)	208 (2.60%)	4464 (55.80%)	2972 (37.15%)

Comme déjà mentionné la zone sensible du processeur LEON3 comprend non seulement les registres de la fenêtre des registres, mais aussi le PC, le nPC, la cache des instructions et la cache de données soit 66400 bits. Le Tableau 3.10 montre les résultats d'une campagne d'injection des fautes, où la limite de temps est établie pour être 5 fois le temps d'exécution, ciblant toute la zone sensible du processeur LEON3. Dans cette campagne ont été injectées 88410 fautes avec l'approche CEU. Le résultat obtenu est un taux d'erreurs global (résultats + timeouts) de 4.08%.

Tableau 3.10 : Résultats de l'injection des fautes sur le code C modifié ciblant toute la zone sensible du processeur LEON3

Zone sensible	# des Fautes Injectées	Résultats Erronés	Timeouts	Fautes Silencieuses
66400 bits	88410	2196 (2.48%)	1415 (1.6%)	84799 (95.92%)

Comme la plupart des erreurs sont dues aux SEUs se produisant dans les variables *T* et *D*, un processeur à trois *cores* a été émulé pour implémenter sur le testeur ASTERICS une architecture TMR (Triple Modular Redundancy) chacun des *cores* exécutant simultanément l'algorithme d'auto-convergence. Le vote a été effectué entre les sorties de chacune des répliques, étant donné que les

erreurs peuvent perturber un, deux, trois ou aucun des cores à la fois. Les résultats à la sortie du TMR peuvent être :

- Erreur : s'il y a deux erreurs, ou une erreur et un timeout.
- Timeout : si deux timeouts se produisent résultant de deux injections dans cores différents.
- Convergence : si l'algorithme d'auto-convergence converge dans au moins l'un des cores, avec un résultat correct.

La Figure 3.20 donne un aperçu de TMR implémenté ainsi que les résultas qui la sortie peut fournir.

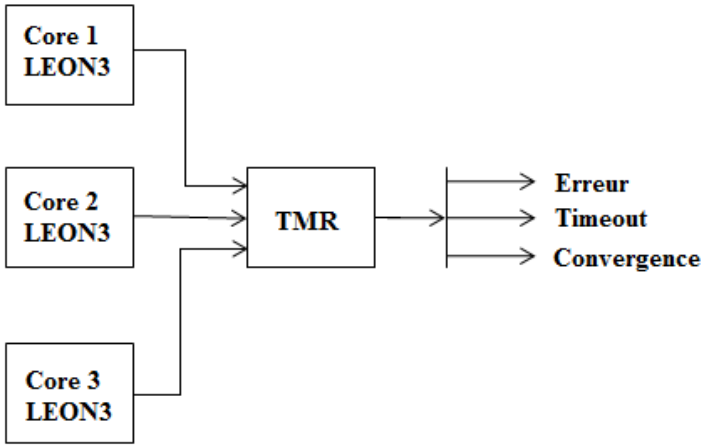


Figure 3.20 - TMR émulé avec 3 cores du processeur LEON3 exécutant le même algorithme d'auto-convergence au même temps

Une campagne d'injection des fautes ciblant les registres de la fenêtre active du LEON3 a été réalisée. La durée d'exécution a été choisie égale à 5 fois le temps d'exécution pour les mêmes raisons évoquées ci-dessus. Le Tableau 3.11 montre les résultats obtenus, mettant en évidence l'efficacité du TMR implémenté. En effet les taux de résultats erronés et des timeouts ont diminué respectivement de 4.45% à 0.65% et de 2.6% à 0.18%, en comparaison avec les résultats de Tableau 3.9. Les résultats obtenus avec les stratégies de tolérance aux fautes présentées dans cette section ont fait l'objet d'une deuxième publication présentée à la conférence RADECS (RAdiation and its Effects on Components and Systems) qui a donné lieu à une publication dans la revue *IEEE Transaction on Nuclear Science* [90].

Tableau 3.11: Résultats de l'injection de fautes sur le TMR

# des Fautes Injectées	Résultats Erronés	Timeouts	Fautes Silencieuses	Convergences
42543	276 (0.65%)	77 (0.18%)	34647 (81.44%)	7543 (17.73%)

Comme évoqué plus haut la zone sensible ne comprend pas seulement les registres de la fenêtre de registres, mais aussi les mémories cache d'instructions et de données et les deux compteurs programme, PC (Program Counter) et nPC (next Program Counter). Les caches d'instruction et de données du processeur LEON3 peuvent être accédés par les instructions *atomic load* et *store unsigned byte*, avec les identifiants spécifiques d'espace d'adressage (*asi codes*), respectivement 0x0a et 0x08. Les deux mémoires cache sont sélectionnées pour être directement mappées, avec 1Kb et 16 mots par ligne.

La méthode CEU permet la lecture d'une adresse de la mémoire cache choisie de façon aléatoire, le masquage avec des données aléatoires pour modifier un seul bit, et ensuite réécrire les données masquées dans le cache et continuer l'exécution du programme avec l'erreur injectée. Un échantillon de ce processus est donnée dans ce qui suit, où "lda" et "sta" sont les instructions arithmétiques *load* et *store*, respectivement. L'instruction "flush" est utilisée pour actualiser le contenu de la mémoire. Les registres %r1, %r2 et %r3 représentent respectivement l'adresse du mot de la cache choisi pour injecter le SEU, les données correspondant à l'adresse et au masque qui devrait être appliqué.

```
flush
lda [%r1] asi, %r2
xor %r2, %r3, %r2
flush
sta %r2, [%r1] asi
```

Le PC et le nPC sont également deux zones sensibles importantes. Ils peuvent conduire soit à des erreurs si le bit perturbé force le PC ou le nPC à manquer ou ré-exécuter un ensemble d'instructions, ou à la perte de la séquence. Dans le processeur étudié, le PC et le nPC sont situés dans les registres locaux %l1 et %l2 de la fenêtre de registres de la routine d'interruption. Donc pour injecter des fautes dans ces deux registres critiques, la même approche CEU peut être utilisée.

Dans le Tableau 3.12 sont présentés les résultats d'une campagne d'injection de fautes afin de vérifier la fonctionnalité de l'algorithme d'auto-convergence modifié, montrant l'efficacité de TMR implémenté sur un processeur LEON3 à trois *cores*. Comme dans les cas précédents, la limite d'exécution est fixée à cinq fois le temps d'exécution nominal. Dans cette campagne, les fautes ont été injectées dans toute la zone sensible accessible par la méthode CEU (ce qui signifie une zone sensible plus grande que celle considérée dans les campagnes d'injection de fautes précédentes), c'est à dire 2075 registres répartis sur la cache de données (1Kb), la cache d'instruction (1Kb), les registres de la fenêtre des registres (25), le PC et le nPC. Les SEUs seront injectés dans un bit aléatoirement choisi du registre sélectionné. Ces expériences conduisent à moins de résultats erronés ainsi que à très peu de timeouts.

Tableau 3.12 : Résultats de l'injection de fautes dans toute la zone sensible de la version trois *cores* du LEON3

# des Fautes Injectées	Résultats Erronés	Timeouts	Fautes Silencieuses	Convergences
100000	85 (0.085%)	15 (0.015%)	98075 (98.075%)	1825 (1.825%)

L'analyse des résultats montre que comme attendu, les erreurs sont la conséquence de deux ou trois injections de SEU dans deux ou trois cores différents chacun à un instant aléatoire. Parmi les 85 erreurs observées (voir Tableau 3.12), 37 provenaient des injections triples et 48 des injections doubles. Dans les Figures 3.21 et 3.22 sont indiquées les répartitions des fautes injectées où DC, IC, PC, nPC et RF représentent les ressources du processeur LEON3. Les figures montrent aussi le type d'erreurs ainsi que le nombre d'erreurs pour chaque type.

Le TMR implémenté est capable de tolérer 100% de fautes injectées dans le cas où un seul SEU est injecté par exécution. Dans le cas de 2 ou 3 fautes par exécution on a des erreurs et le TMR ne peut faire face a ce genre de fautes puisque les erreurs sont dans deux or trois cores différents du LEON3, c'est à dire le TMR ne peut pas détecter des erreurs dans deux ou trois cores du LEON3 à la fois puisqu'il prend deux votes parmi trois. Pour détecter des fautes ayant lieu dans deux cores il faut un QMR (Quintuple Modular Redundancy) qui prend trois votes parmi cinq.

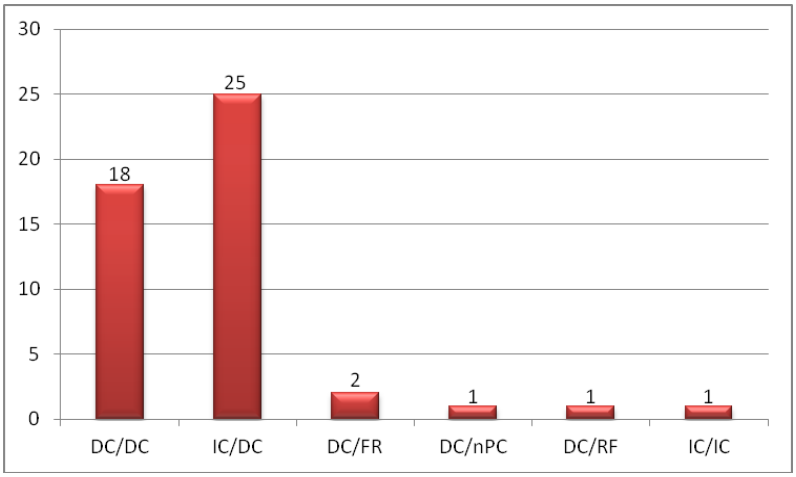


Figure 3.21 - Distribution des erreurs résultant des injections doubles

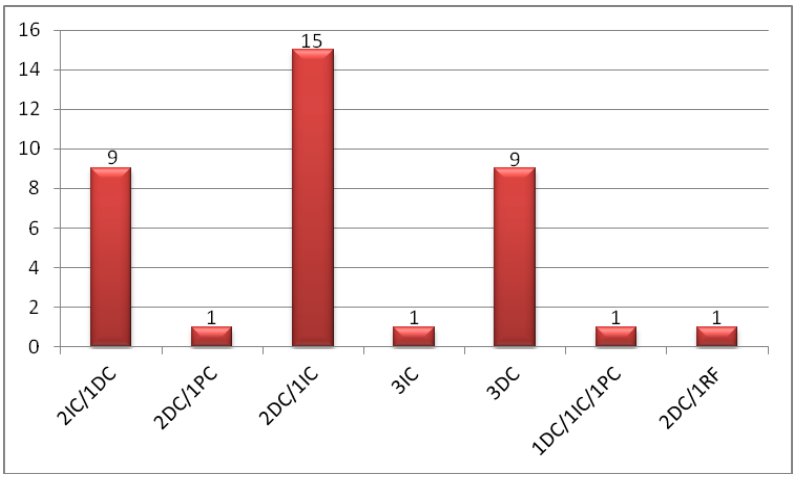


Figure 3.22 - Distribution des erreurs résultant des injection triples

Les résultats obtenus das ces campagnes de test peuvent être comparés aussi à ceux issus des campagnes avec les stratégies adoptées dans la sous-section 3.4.3 afin de valider la tolérance aux SEUs de l'algorithme d'auto-convergence. Les résultats obtenus montrent une réduction des résultats erronés et des timeouts avec le TMR implémenté dans cette section. Le TMR apporte un gain total de 4.52% (résultats erronés + timeouts) en comparaison avec les modifications en langage d'assemblage de l'algorithme d'auto-convergence.

Étant donné que les timeouts et les erreurs ne dépassent pas 0.1% du total des SEUs injectés, une solution pour les éliminer consiste à utiliser un *watchdog* et redémarrer la simulation en cas d'erreurs ou timeouts.

La technique implémentée considère une aproche traditionnelle de tolérance aux fautes (TMR) et l'auto-convergence, l'efficacité de cette technique est expliquée par le fait que les premières méthodes tentent de masquer l'apparition d'erreurs et ainsi éviter la faute, tandis que l'autostabilisation assure la récupération si une faute transitoire se produit. De cette façon, l'auto-convergence offre une approche complémentaire à des méthodes de tolérance aux fautes. Si l'on considère, par exemple, la communication entre les *cores* d'un processeur *multicore*, où un *core* veut envoyer un message à un autre *core*, le message est envoyé trois fois (sans auto-convergence) et passent par un TMR, c'est à dire le message est transmis trois fois, et le résultat est déterminé par vote, et en fonction du résultat le *core* récepteur adopte la décision du cas. Dans cette technique le masquage de fautes assure la réponse correcte même en présence de fautes, si la faute ne se manifeste pas comme une erreur. Autrement, si le système passe à un état erroné, donc les erreurs doivent être détectées et récupérées. Dans ce contexte, considérant le TMR avec l'auto-convergence est possible d'admettre que le système soit dans un état erroné pendant un certain temps, le temps nécessaire à la détection de l'erreur et à la récupération d'un état libre de fautes.

3.5 Conclusion

Dans ce chapitre a été étudié l'impact des fautes de type SEU sur un algorithme d'auto-convergence, algorithme utilisé dans les systèmes distribués afin de garantir la convergence des données transmises, même en cas d'erreurs. Les résultats obtenus dans les premières campagnes d'injection de fautes effectués ont mis en évidence une certaine sensibilité aux SEUs de la version sans durcissement de l'algorithme étudié soit au niveau C code ou d'assemblage. Après cette étape, une version améliorée de l'algorithme d'auto-convergence, incluant des modifications appropriées et la technique de tolérance aux fautes TMR (Triple Modular Redundancy), a été développée et testée par injection de fautes. Les résultats de ces campagnes d'injection de fautes ont mis en évidence l'efficacité du TMR implémenté avec l'auto-convergence de l'algorithme, les résultats ont montré un gain total significatif de 17.82% (ce résultat est la différence entre la somme des taux des résultats erronés et des timeouts des campagnes utilisant l'algorithme sans durcissement et l'algorithme avec durcissement).

La suite de ce manuscrit décrira la dernière partie de l'étude sur l'auto-convergence, dans laquelle sera considéré un circuit auto-convergeant dédié implémenté dans un FPGA. La plateforme de test utilisée est ASTERICS, cependant l'injection de fautes sera effectuée au niveau RTL (Register Transfer Level) et des campagnes d'injection de fautes seront effectuées pour obtenir des résultats sur la robustesse intrinsèque de cette implémentation face aux SEUs.

Chapitre 4. Implémentation et test d'une version hardware de l'algorithme d'auto-convergence

Dans ce chapitre est décrite une implémentation d'un circuit self-convergeant dans un FPGA. Une méthode d'injection de fautes au niveau circuit sera utilisée pour obtenir des résultats sur la robustesse intrinsèque de l'implémentation étudiée face aux soft erreurs résultant des effets des radiations. Les résultats obtenus seront comparés à ceux obtenus par les campagnes d'injection de fautes réalisés sur le processeur LEON3 exécutant la version logicielle de l'algorithme d'auto-convergence étudié, lorsque des fautes sont injectées dans ses variables.

4.1 Implémentation dans un FPGA de l'algorithme d'auto-convergence

Comme indiqué dans les chapitres précédents l'algorithme d'auto-convergence est un algorithme tolérant aux fautes qui récupère son comportement normal, même si il est mal initialisé ou perturbé lors de son exécution. L'algorithme présenté dans le chapitre 2.7 est bien connu et largement utilisé dans les domaines liés à l'informatique distribuée: le plus court chemin, les réseaux des capteurs, les réseaux informatiques, etc. Dans cette section est présentée l'implémentation sur FPGA d'un circuit dédié, l'algorithme d'auto-convergence, qui se basera sur deux types de cellules. La première cellule, appelée nœud de base (*basic_node*), est responsable de l'addition de T_{ij} et D_j, alors que la deuxième cellule, appelée nœud de comparaison (*cmp_node*), est responsable de la comparaison de deux valeurs, afin de fournir en sortie le minimum entre eux. Les codes Verilog de l'algorithme auto-convergeant implémenté sont donnés en détails dans l'ANNEXE J.

4.1.1 Le nœud de base

Avant d'entrer dans les détails de l'implémentation, il est important de mentionner que les éléments des deux matrices *T* (matrice d'entrées) et *D* (matrice de sortie) sont considérées comme des données de 12 bits. La raison de choisir cette largeur est de pouvoir s'adapter à la longueur des arêtes reliant deux nœuds distincts.

Comme mentionné précédemment le nœud de base a pour but l'addition des éléments T_{ij} et D_j. Dans la Figure 4.1 est donnée l'architecture du nœud de base. La première étape qui effectue le processus d'addition selon l'équation (2.2) présentée avec plus de détails dans le chapitre 2 est montrée ci-dessous:

$$D_0 = 0;\ \ D_i = \min\{T_{ij} + D_j\}$$

Les éléments T_{ij} et D_j sont ajoutés (adder) et leur résultat est stocké dans un registre appelé PIPO (Parallel-In Parallell-Out), i.e., entrée-parallèle et sortie-parallèle. Le signal de réinitialisation actif bas (rst_n) est également enregistré dans une flip-flop de type D (DFF) pour maintenir la synchronisation entre les nœuds. Les sorties de nœud de base sont *out_o* (résultat de sortie) et *rst_o* (sortie reset).

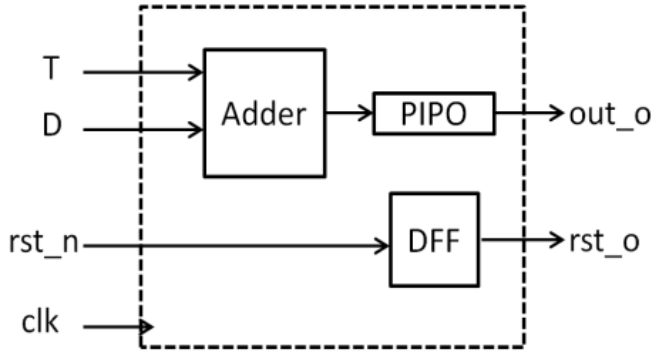


Figure 4.1- Architecture du nœud de base (*basic_node*)

4.1.2 Le nœud de comparaison

Le nœud comparateur est responsable de la sélection de la valeur minimum de deux entrées afin d'implémenter complètement l'équation (2.2). Ces deux entrées correspondent aux sorties *out_o* de chaque nœud de base. L'architecture de ce nœud est basée sur un soustracteur qui permet de détecter à travers du signal *Bout* (*borrow out*) si l'entrée *I1* 12-bits est inférieure à l'entrée *I2*. Dans la Figure 4.2 est donnée l'architecture du nœud de comparaison *cmp_node*. Le signal *Bout* représente la ligne de sélection d'un multiplexeur 2-vers-1. La valeur minimale entre les deux entrées est mémorisée dans un registre PIPO et fourni à la sortie *min_o* en 12 bits. Pour faire face aux problèmes de synchronisation, le signal de réinitialisation actif bas, *rst_n* est stocké dans un DFF (D flip-flop) et est fourni à la sortie comme *rst_o*.

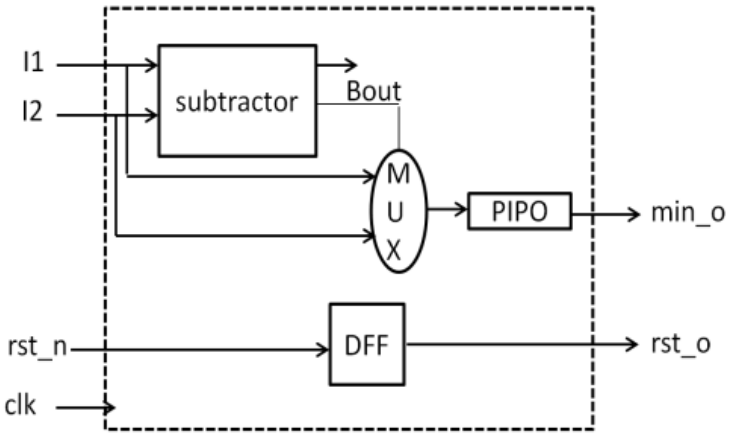


Figure 4.2 - L'architecture du nœud de comparaison (*cmp-node*)

4.1.3 L'architecture d'une ligne

L'implémentation proposée de l'algorithme d'auto-convergence consiste à permettre au réseau (N x N nœuds de base) se développer d'une mainière modulaire. Afin de calculer la nouvelle valeur de D_i dans l'équation (2.2), il faut ajouter sa valeur précédente, D_j, avec tous les éléments de la ligne i de la matrice T dont la taille est N x N. Ainsi, le résultat est le minimum de l'ensemble des valeurs additionnées.

L'opération pour avoir la nouvelle valeur de D_i peut être effectuée en mettant en place une ligne de N nœuds de base (*basic_nodes*), dans laquelle à chaque paire de nœuds de base voisins est ajouté un nœud de comparaison (*cmp_node*). Dans la Figure 4.3 est montrée l'architecture des lignes de 2 et 4 nœuds.

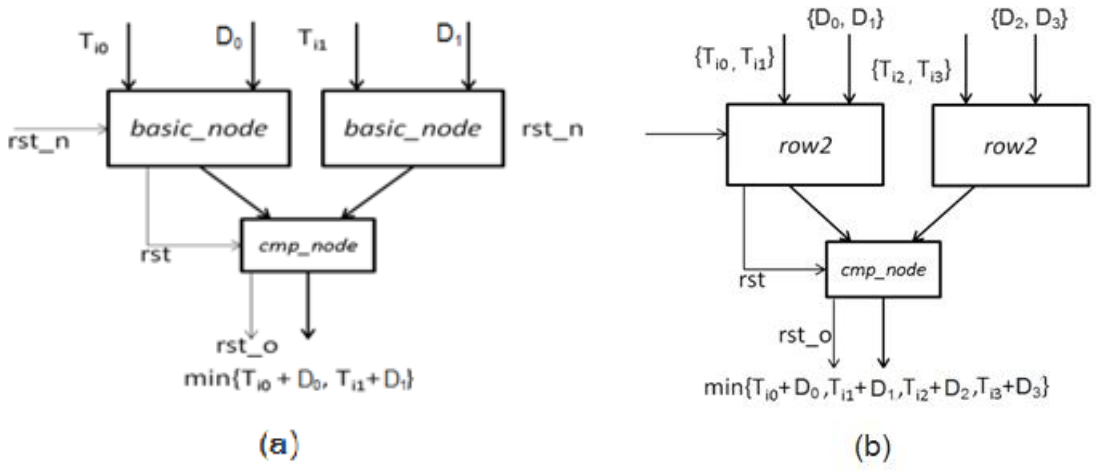


Figure 4.3 - L'architecture de ligne: (a) N = 2 (b) N = 4

La Figure 4.4 montre l'architecture d'une ligne de l'ordre 8 et de l'ordre 16. On peut voir aussi le calcul de la valeur minimum de D_i pour chaque i.

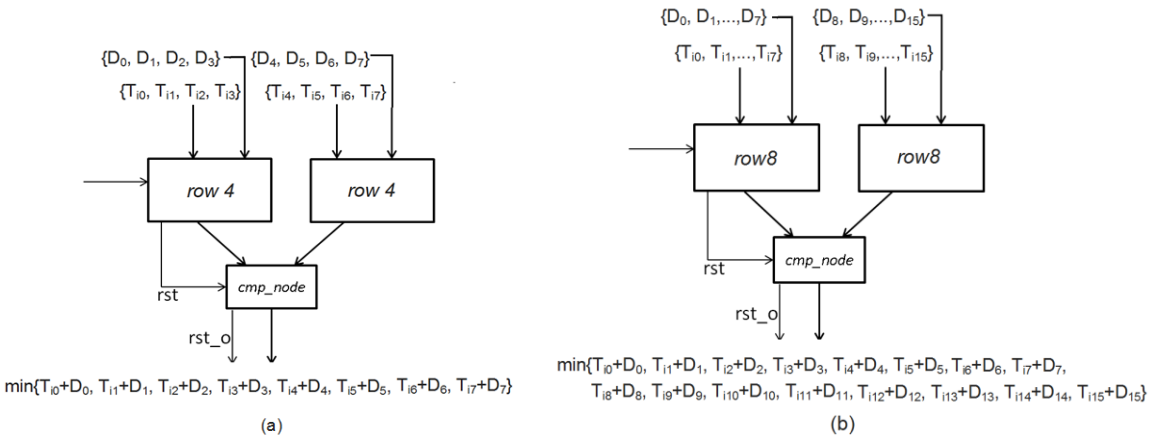


Figure 4.4 - L'architecture de ligne: (a) N = 8 (b) N = 16

4.1.4 L'architecture *mesh*

Le processus d'auto-convergence est arrêté lorsque la matrice *D* calculée est exactement égale à sa valeur précédente, ainsi on peut dire que l'algorithme a convergé. Il faut mentionner que la matrice *D* est initialisée à zéro et la matrice *T* est stockée dans une mémoire RAM présente dans le FPGA.

Pour contrôler le processus entier, une machine à états finis (FSM) est implémentée ayant pour rôle: l'obtention des valeurs de la matrice *T*, la réinitialison des nœuds, la vérification de la matrice *D* et l'envoie de sa valeur au réseau jusqu'à ce que la condition d'arrêt soit satisfaite. La Figure 4.5 présente l'architecture *mesh* 8 x 8 connectée à la machine d'état et aux deux matrices *T* et *D*. Le Tableau 4.1 fournit des détails sur les ressources matérielles utilisées pour implémenter un réseau 16 x 16 sur un FPGA Virtex IV (XC4VFX40).

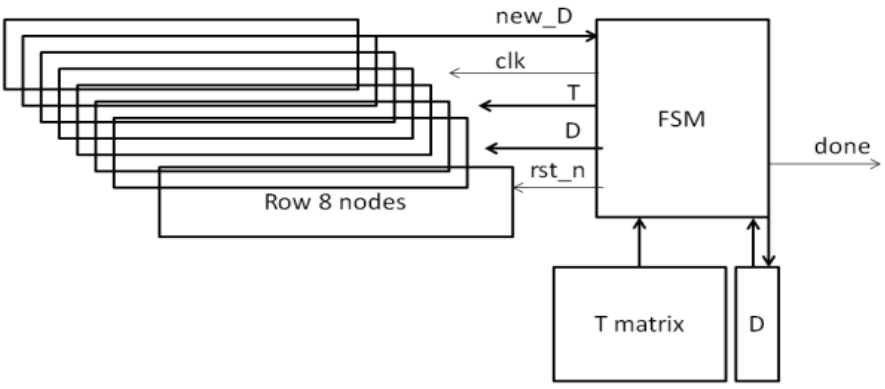


Figure 4.5 - L'architecture mesh 8 x 8

4.2 L'environnement d'injection des fautes

Cette section a pour but donner un aperçu de l'environnement d'injection de fautes au niveau RTL, utilisé pour évaluer la tolérance aux fautes du circuit HDL implémentant l'algoithme d'auto-convergence. La plateforme ASTERICS décrite dans le chapitre 3.3.1, a été utilisée pour étudier les effets SEU sur le circuit self-convergeant implémenté dans un FPGA.

4.2.1 L'approche NETFI d'injection de fautes

L'approche d'injection de fautes NETFI (NETlist Fault Injection), au cœur des travaux d'une thèse précédente de TIMA [7] consiste à injecter des fautes dans un circuit au niveau RTL (Register Transfer Level) implémenté sur un FPGA. Cette approche permet l'injection de fautes du type SEU dans un circuit d'une manière entièrement automatisée sur n'importe quel bloc de mémoire vive (BRAM) ainsi que sur les cellules mémoire du design étudiée. Cette approche est basée sur une interaction software/hardware. La partie software est seulement utilisée sur l'ordinateur de l'utilisateur qui exécute l'outil responsable du control des campagnes d'injection de fautes. La partie matérielle est implémentée sur la plateforme ASTERICS, qui reçoit les commandes de l'ordinateur, et supervise le processus d'injection de fautes.

La méthode NETFI permet d'effectuer l'injection de fautes dans un cycle d'horloge, de façon aléatoire dans le temps et le lieu, en émulant ainsi l'effet des fautes du type SEU dans le circuit étudié. L'idée de cette approche est de modifier de manière non intrusive pour le design étudié, les cellules sensibles de la bibliothèque *built-in* du FPGA (Xilinx) qui seront utilisées par la netlist après synthèse du circuit cible au niveau RTL, permettant ainsi l'injection de fautes. L'automatisation complète des étapes à partir de la synthèse et jusqu'à l'injection de fautes est aussi une caractéristique significative de l'approche d'injection de fautes NETFI de sorte que pour tout circuit HDL qui fonctionne correctement sur le FPGA l'injection de fautes peut être réalisée avec un minimum de temps et d'effort. Les caractéristiques de l'approche NETFI sont résumées ci-dessous :

- Les fautes sont injectées au niveau netlist.
- L'émulation automatique des fautes.
- Applicable sur les dispositifs de Xilinx.
- Manipulation de la netlist en remplaçant la bibliothèque buit-in de Xilinx par d'autres ayant la même fonctionnalité.
- Peut injecter de fautes de type SEU, SET et Stuck-at, mais cette thèse s'est intéressé seulement par les fautes de type SEU.
- Pas des restrictions concernant la capacité du FPGA (pas des problèmes de routage).

La Figure 4.6 montre le schéma initial de l'approche NETFI. Le code HDL du circuit cible est synthétisé de manière à extraire la netlist. Cette synthèse est effectuée par un outil appelé *Synplify Pro* qui exécute un script avec toutes les options de synthèse: le choix du FPGA, la génération de la netlist, etc. Le Chipset FPGA de la plateforme ASTERICS a été choisi comme FPGA de synthèse pour générer la netlist du circuit cible. Après synthèse, la netlist peut contenir différents types des cellules built-in du FPGA tels que les flip-flops: FD (flip-flop type D), FDC (D flip-flop avec un signal *clear* ou *reset*), FDE (D flip-flop avec le signal enable) les mémoires BRAMs et les circuits combinatoires mappés dans les tableaux LUTs (look-up tables). Ces composants sont tous modifiés en ajoutant un signal injection *INJ* qui permet d'injecter des fautes à un instant aléatoirement choisi.

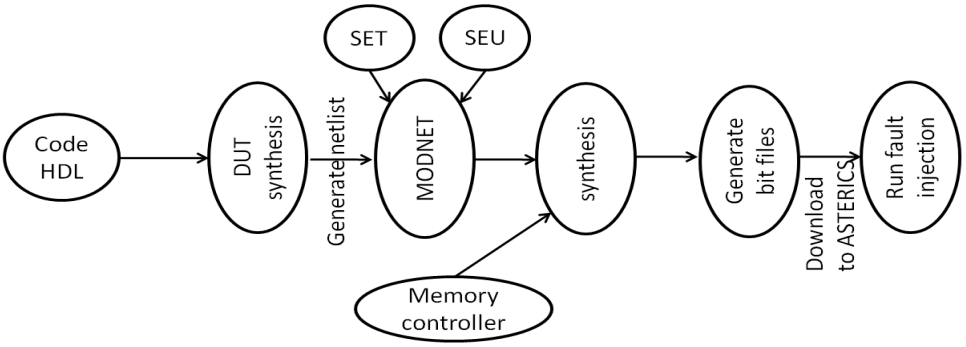


Figure 4.6 - Schéma de l'approche NETFI

La netlist après la synthèse est manipulée par un outil appelé MODNET (MODify NETlist) qui sélectionne le type de faute qui sera injecté et délivre une netlist modifié. La nouvelle netlist constitue un sous-module complet du circuit RTL qui contient un contrôleur de mémoire pour stocker les erreurs détectées dans la mémoire statique (SRAM) de la plateforme ASTERICS, et un superviseur qui contrôle le processus d'injection de fautes. L'étape de placement et routage (*place and route*) suit la re-synthèse afin de configurer le Virtex IV Chipset FPGA de l'ASTERICS et exécuter les expériences d'injection de fautes. Une première exécution, appelée *Golden Run*, est lancée pour obtenir les paramètres nécessaires aux tests : les résultats de référence et la durée d'exécution.

L'outil NETFI est configurable et son entrée est un fichier RTL testé dans l'environnement d'ASTERICS, en d'autres termes le fichier RTL peut être implémenté comme un sous-module du circuit de configuration de la plateforme. Son accès à l'environnement ASTERICS permet aussi le débogage du code RTL. L'approche NETFI est automatisée sous la plateforme *Windows* à travers d'une application (*console prompt*) qui effectue différentes commandes. Ces commandes sont décrites en détails dans [7].

4.2.2 Le circuit auto-convergeant après NETFI

La méthode d'injection de fautes NETFI décrite auparavant a été utilisé pour l'étude de la sensibilité par rapport aux SEUs d'un circuit auto-convergeant. Dans ces travaux on a implémenté dans un FPGA un circuit auto-convergeant qui a pour but la recherche des plus courts chemins dans un graphe (voir section 4.1). Ce circuit a été implémenté sur le *Chipset* FPGA de la plateforme ASTERICS pour effectuer l'injection de fautes. Le Tableau 4.1 fournit des détails sur les ressources matérielles utilisées pour implémenter un réseau 16 x 16 sur un FPGA Virtex IV (XC4VFX40).

Tableau 4.1: Ressources utilisées pour implémenter un circuit auto-convergeant 16 x 16 sur un FPGA Virtex IV

L'utilisation logique	**Utilisé**	**Disponible**	**Utilisation**
Slice	5950	18624	31%
Slice Flip-Flops	7603	37248	20%
4 entrées LUTs	9041	37248	24%

La méthode NETFI, lance la synthèse du code Verilog (les codes Verilog du circuit cible sont montrés dans l'ANNEXE J) du circuit auto-convergeant pour obtenir la netlist qui sera modifiée pour permettre l'injection de fautes. Cette synthèse choisi le *Chipset* FPGA de testeur ASTERICS pour générer la netlist du circuit cible, ceci avant la modification NETFI. L'analyse de la netlist obtenue après synthèse a permis d'identifier 7041 flip-flops, modules built-in de Xilinx. Ces 7041 flip-flops ont été modifiés en ajoutant un signal injection "*INJ"* qui permet d'injecter des fautes à un instant aléatoirement choisi. Les modifications de ces modules flip-flops sont montrés dans l'ANNEXE L.

4.2.3 L'injection de SEU avec la méthode NETFI

Dans ce qui suit sont montrées les modifications des cellules sensibles de la bibliothèque built-in du FPGA pour pouvoir faire l'injection de fautes. Parmi les cellules contenu dans la netlist après la synthèse une catégorie principale est ciblé :

- **Les fip-flops:** la modification de toutes les flip-flops est nécessaire pour pouvoir changer leurs contenus à un instant aléatoire et donc pour faire en sorte que l'émulation de fautes de type SEUs au niveau netlist puisse avoir lieu pour le circuit choisi. Pour ce faire les flip-flops sont classés en deux types : les flip-flops sans signal d'activation et ceux avec signal d'activation. Les premières flip-flops ont toujours une donnée active sur leur sortie et la

modification est faite en ajoutant le signal d'injection *INJ* pour pertuber leur contenu pendant un seul cycle d'horloge et ainsi permettre l'injection de fautes en choisissant d'écrire dans la flip-flop considérée soit la donnée entrante ou la donnée entrante inversée. Pour ce qui est des flip-flops avec un signal d'activation, il doit être actif pour permettre l'injection de fautes en ajoutant le signal *INJ*. L'écriture dans la flip-flop sera faite si le signal *chip-enable* (CE) ou le signal *INJ* sont actifs. En effet, si les signaux *INJ* et *CE* sont actifs la donnée *D* inversée sera écrite dans la flip-flop; mais dans le cas où le signal *INJ* soit actif et *CE* inactif, la sortie *Q* inversée est réécrite. Dans le cas òu le signal *INJ* est inactif la flip-flop garde son fonctionnement original. Les Figures 4.7 et 4.8 montrent les schémas avec les modifications de la D flip-flop sans et avec le signal d'activation (CE), respectivement.

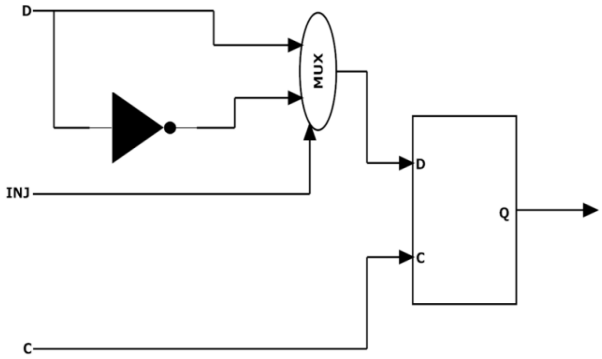


Figure 4.7 - Modification de la D flip-flop sans le signal d'activation pour l'injection de SEU avec la méthode NETFI

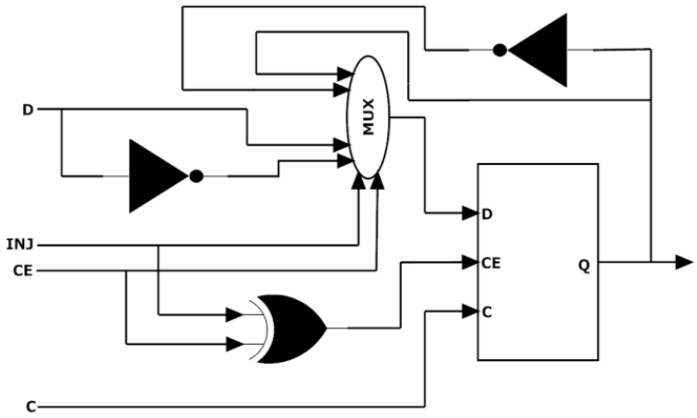


Figure 4.8 - Modification de la D flip-flop avec le signal d'activation CE pour l'injection de SEU avec la méthode NETFI

4.3 Résultats expérimentaux obtenus par la méthode NETFI appliquée au circuit auto-convergeant

La méthode NETFI peut être utilisée pour l'évaluation de la sensibilité des circuits face aux SEUs, ainsi que pour la validation des techniques de tolérance aux fautes [7][95].

L'algorithme étudié est le même que celui utilisé pour l'injection de fautes avec la méthode CEU, décrite dans la section 3.1. Cet algorithme a pour but trouver le plus court chemin d'un graphe de 16 noeuds. La dimension de la matrice *T* est donc 16 x 16 pendant que la matrice *D* a une dimension de 16 x 1. Notez que ce benchmark a été également exécuté, dans sa version logicielle, par un processeur LEON3 (voir sous-section 3.4.2).

Plusieurs *runs*, au cours de laquelle des très nombreux SEUs ont été injectés de façon aléatoire dans le temps et le lieu, ont été réalisés. Il est important de noter que dans chacun de ces runs, un seul SEU par exécution a été injecté dans les cellules mémoire cible sélectionnées du circuit implémentant l'algorithme d'auto-convergence. En d'autres termes, le contenu de la flip-flop choisie est perturbé durant le temps d'exécution de l'algorithme étudié et à un cycle choisi au hasard.

Les conséquences des fautes injectées sont classées de la même façon que dans le chapitre 3 :

- Fautes silencieuses : la faute injectée n'a aucun effet.
- Résultats erronés : la sortie de l'algorithme d'auto-convergence n'est pas celui attendue.
- Timeouts : l'exécution n'arrive pas à terme.
- Convergence : le circuit fournit des résultats corrects après la limite d'exécution prévue.

Dans nos expériences, les fautes SEU ont été injectées à des instants choisis dans la durée nominale du *benchmark* exécuté. Comme indiqué dans le Tableau 4.2, plusieurs campagnes d'injection de fautes avec différentes limites d'exécution ont été effectuées. Comme on peut voir dans la dernière colonne du Tableau 4.2, ces limites d'exécution sont donnés en termes de "x fois la durée d'exécution nominale" (c'est à dire l'exécution sans injection de fautes) et ont les mêmes valeurs que celles utilisées dans les campagnes d'injection de fautes pour le LEON3 présentées dans le chapitre 3. Ces campagnes avec différentes limites d'exécution ont pour but obtenir des évidences sur l'impact de la limite d'exécution sur le nombre et le type d'erreurs.

Dans le Tableau 4.2 sont résumés les résultats des campagnes d'injection de fautes effectuées. Ces résultats montrent que 0,18% de fautes injectées échappent au circuit implémentant l'algorithme d'auto-convergence les échappe. Si on compare avec la version logicielle de l'algorithme d'auto-convergence présenté dans la section 3.4.2, les résultats montrent que les deux implémentations stabilisent lorsque que la limite d'exécution est supérieure ou égale à 5 fois le temps de la durée

nominale. Les résultats de ces campagnes d'injection de fautes sur le circuit auto-convergeant étudié ont donné lieu à un quatrième article présenté à la conférence ICM'2013 (International Conference on Microelectronics) [96].

L'une des principales contributions d'implémenter l'algorithme d'auto-convergence sur un circuit hardware dédié est de montrer sa robustesse et son efficacité face aux SEUs. La durée d'exécution nominale de l'algorithme implémenté est de 310 cycles ce qui donne 31 µs pour une horloge de 10 MHz. Ce temps de traitement est 10^5 fois plus rapide que celui de la version logicielle présentée dans la section 3.4.2. Une autre contribution importante est la diminution significative de la moyenne globale des taux d'erreurs (résultats erronés + timeouts). Par exemple, dans le troisième essai, le taux global d'erreurs diminue, passant de 17.92% (voir Tableau 3.2 dans la section 3.4.2) pour la version logicielle, à 0.18% pour celle hardware.

Les résultats obtenus de ces campagnes sur le circuit auto-convergeant peuvent être comparés aussi aux résultats obtenus avec les campagnes d'injection de fautes sur le processeur trois *cores* émulé pour implémenter un TMR (résultats présentés dans le chapitre 3), une fois que la sortie est la même. Considérant le test 3 dans le Tableau 4.2, où la durée d'exécution est égale à 5 fois la fin attendue, à cette limite le taux de résultats erronés et des timeouts se stabilisent, le taux d'erreurs sont réduits à environ la moitié, passant de 0.18% pour la version circuit HDL auto-convergeant et à 0.1% pour le TMR implémenté sur un processeur trois cores. Il est important de noter que l'algorithme d'auto-convergence exécuté par chacun des cores a été modifié (un opérateur modulo «%» supplémentaire a été ajouté avec chaque appel des entrées et des sorties; à chaque variable du programme a été attribué um registre local dans la fenêtre des registres et les variables booléennes sont initialisées avec une valeur autre que «1» pour éviter les timeouts produits par les SEUs affectant ces variables) pour atteindre des taux d'erreurs plus faibles, et donc montrer l'efficacité du TMR implémenté.

Tableau 4.2: Résultats des campagnes d'injection de fautes sur le circuit auto-convergeant implémenté en RTL dans un FPGA

Test	**# des Fautes Injectées**	**# Résultats Erronés**	**#Timeouts**	**# Fautes Silencieuses**	**#Convergences**	**Limite d'exécution**
1	20145	20 (0.10%)	42 (0.21%)	19871 (98.64%)	212 (1.05%)	1.5
2	20313	24 (0.12%)	14 (0.07%)	20033 (98,62%)	242 (1.19%)	3
3	20016	29 (0.14%)	9 (0.04%)	19737 (98.61%)	241 (1.20%)	5
4	22013	34 (0.15%)	6 (0.03%)	21722 (98.68%)	251 (1.14 %)	16

NETFI identifie 7041 flip-flops qui sont les cibles des SEU pour les campagnes d'injection de fautes. Le FSM (Finite State Machine) et la matrice T (entrée du système) occupent environ 614 de ces flip-flops.

Pour le troisième essai où la limite d'exécution est égale à 5 fois la durée d'exécution nominale, les analyses détaillées de la répartition de SEU affectant le résultat final de l'algorithme sont résumés dans les Figures 4.9, 4.10 et 4.11. La raison du choix de ce test est que, comme indiqué précédemment, à cette limite d'exécution le taux d'erreurs et de timeouts stabilise.

La Figure 4.10 montre que des SEU injectés dans différentes adresses du circuit cible peuvent être tolérées et que le circuit converge toujours vers une sortie correcte après quelques itérations supplémentaires.

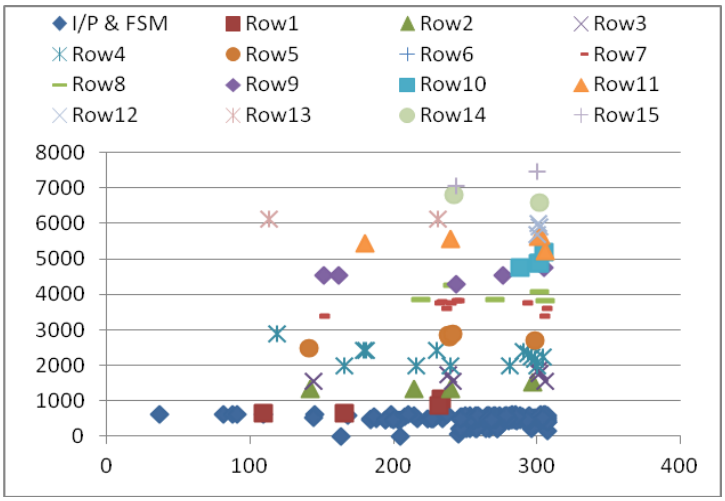


Figure 4.9 - Répartition des SEU provoquant convergence de l'algorithme: adresse en fonction du temps (limite d'exécution = 5 x la durée nominale)

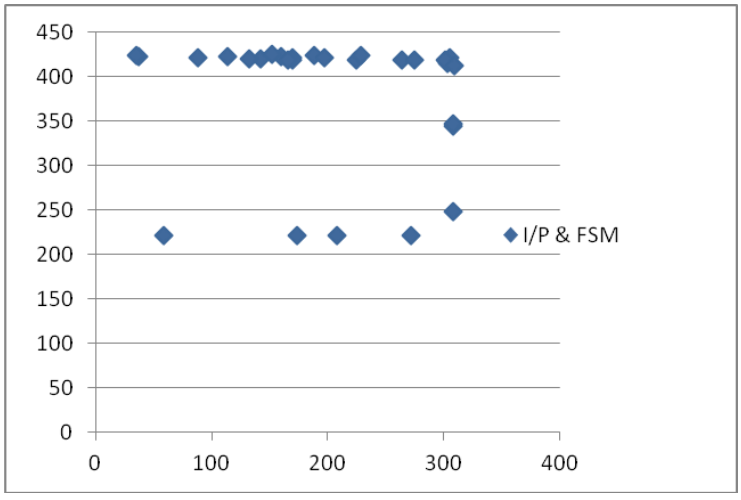


Figure 4.10 - Répartition des SEU provoquant des résultats erronés de l'algorithme: adresse versus. temps (limite d'exécution = 5 x la durée nominale)

Les Figures 4.10 et 4.11 montrent les fautes qui échappent à la méthode étudiée et qui provoquent des erreurs ou des timeouts.

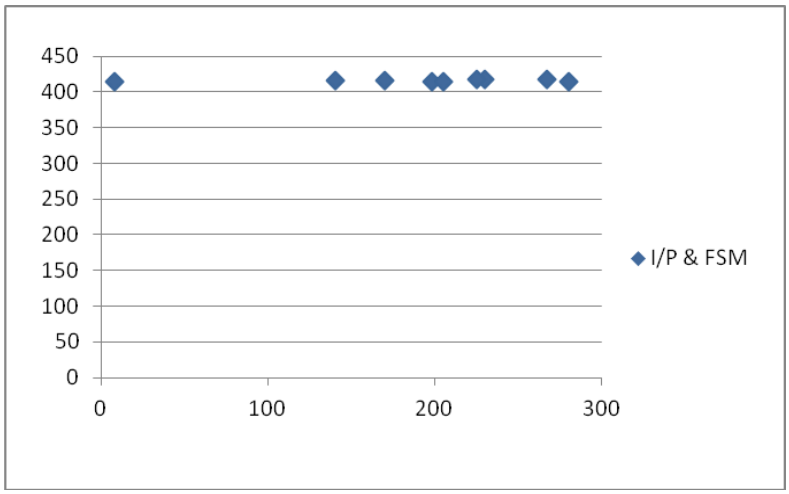


Figure 4.11 - Répartition des SEU provoquant des timeouts de l'algorithme: adresse versus temps (limite d'exécution = 5 x la durée nominale)

4.3.1 Les fautes non détectées par la méthode

L'analyse approfondie des résultats sur la zone sensible de circuit auto-convergeant montre que toutes les erreurs se produisent lorsque les fautes injectées ciblent soit la région où la machine à états du système est implémentée soit la matrice *T*, qu'est l'entrée du système. Dans les Figures 4.10 et 4.11 sont montrées les distributions des SEUs provoquant des erreurs et timeouts du circuit auto-convergeant étudié. Lorsque les données de la matrice *T* sont perturbées le circuit converge mais pas au bon résultat, produisant ainsi des erreurs. Si la machine à états est perturbée, les conséquences de cette perturbation seront des timeouts et résultats erronés, une fois que, comme déjà dit, la machine à états contrôle le processus entier: alors la condition d'arrêt (valeur courante de la matrice *D* est égale à sa valeur précédente) peut jamais arriver si la valeur courante de la matrice *D* est toujours différente de sa valeur précédente, ainsi on reste dans la boucle et on a pour résultat la perte de séquencement qui peut conduire à un timeout. Dans le cas où la valeur précédente de *D* est perturbée et sa valeur est égale à la valeur courante de *D* on sort de la boucle avant de la finir ce qui conduit à des résultats erronés.

Pour faire face aux timeouts détectés, un chien de garde peut être utilisé et le hardware peut être remis à zéro après un délai spécifique. Dans tous les cas, le taux d'erreurs peut être diminué si la technique TMR (Triple Modular Redundancy) est appliquée à la FSM (machine à états) et à la matrice *T* ce qui permet d'assurer la complète robustesse de l'algorithme d'auto-convergence implémenté sur hardware.

4.3.2 Synthèse des conséquences des résultas des campagnes de test

Le but dans cette sous-section est de résumer les résultats des campagnes d'injection de fautes présentées dans cette thèse. Plusieurs campagnes d'injection de SEUs dans un processeur LEON3 exécutant un algorithme d'auto-convergence et dans un circuit auto-convergeant dédié implémentés dans un FPGA ont été effectuées. Les résultats obtenus confirment la capacité de cet algorithme à fournir des résultats corrects en présence de fautes affectant les données et les registres utilisées par l'algorithme. En plus, il est montré que la durée d'exécution peut cacher des erreurs qui ont été considérées comme timeouts dans des campagnes ayant un 'timeout limit' plus petit. Puis une version en langage assembleur de l'algorithme a été considérée et a permis d'obtenir une pathologie des erreurs qui échappent aux capacités de tolérance aux fautes de cet algorithme. Pour les fautes qui ont comme conséquence des résultats erronés, un cas est trouvé et concerne toutes les fautes qui provoquent la fin de la boucle avant le temps nécessaire pour terminer l'exécution de l'application. Dans le cas où les fautes ont comme conséquence des délais d'attente, l'algorithme peut entrer dans une boucle infinie. Les résultats avec une version incluant différentes stratégies, modifications logicielles et l'implémentation d'un TMR implémenté dans un processeur LEON3 trois *cores* ont montré que le TMR est capable de tolérer 100% de fautes injectées. Une version tripliquée de l'algorithme pourrait être considérée comme stratégie pour améliorer les capacités de tolérance aux fautes, cependant cette stratégie a un impact sur le temps d'exécution, c'est à dire une augmentation considerable du temps d'exécution quand on considère chacune des répliques exécutées par un seul *core*, raison pour laquelle cette stratégie n'ont pas été pris en compte dans ces travaux. Enfin, les résultats obtenus sur le modèle RTL d'un circuit dédié implémentant l'algorithme d'auto-convergence ont montré une amélioration significative de la robustesse de l'algorithme en comparaison avec sa version logicielle (version originale). L'analyse de ces résultats a mis en évidence que les erreurs sont les conséquences des SEUs pertubant la matrice *T* et la machine à états, donc une stratégie pour améliorer la robustesse face aux SEUs de ce circuit est l'implémentation d'un TMR sur la FSM (machine à états) et la matrice *T*, ainsi on peut assurer la complète robustesse de cirucuit auto-convergeant. En outre, le temps d'exécution nominal de l'algorithme exécuté par le circuit dédié est 10^5 fois plus rapide que celui de sa version logicielle exécutée par un processeur.

4.4 Conclusion

Dans ce chapitre a été exploré l'impact des SEUs sur un type particulier d'algorithmes, dits auto-convergents, implémentés dans une version hardware dans un FPGA Virtex IV. Les résultats obtenus ont confirmé la capacité de fournir des résultats corrects en présence de fautes affectant les cellules mémoire et les données utilisées par l'algorithme. Ces résultats ont été analysés et comparés

avec ceux obtenus par injection de fautes sur la version logicielle de l'algorithme mettant en évidence une amélioration significative de la robustesse de l'algorithme et une comparaison est également faite avec les résultats obtenus avec ceux issus de l'architecture TMR implémentée sur un processeur trois *cores* étudiée dans le chapitre 3.

Chapitre 5. Conclusions Générales et Perspectives

Les progrès permanents des technologies de fabrication des circuits intégrés permettent l'augmentation de la densité d'intégration des transistors de plus en plus petits dans une seule puce. Due à ce fait il devient possible concevoir des processeurs avec plusieurs *cores* qui au début ont été conçus pour atténuer les limitations physiques, comme la température, des processeurs *singlecore*. Dans le monde numérique d'aujourd'hui, les demandes des complexes simulations 3D, le streaming des fichiers multimédias, des niveaux supplémentaires de sécurité, des interfaces utilisateur plus sophistiqués, de grandes bases de données et le grand nombre des internautes demandent toujours plus de puissance de traitement. Toutefois, en augmentant le nombre de cores sur une seule puce pour avoir une puissance de traitement plus élevée, des défis se présentent avec la mémoire et la cohérence de la mémoire cache, ainsi que la communication entre les *cores*. Donc dans ce contexte il est souhaitable que la communication soit fiable et robuste : les algorithmes dits auto-convergeants offrent la possibilité de garantir le transfert correct des messages entres les *cores*.

Cette thèse a présenté une étude de la fiabilité d'un algorithme d'auto-convergence explorant la robustesse/sensibilité intrinsèque de cet algorithme face aux fautes de type SEU (Single Event Upset) provoquées les particules énergetiques présentes dans l'environnement dans lequel le circuit opérera. L'algorithme d'auto-convergence présente la propriété d'auto-satabilisation qui assure que le système, indépendamment de l'état initial, convergera vers une configuration légitime dans un temps fini, et finira pour récupérer un comportement correct si aucune autre faute ne se produit. Une contribution de ces recherches est la prise en compte des fautes de type SEU (Single Event Upset) comme menace pour les algorithmes d'auto-convergence. Ces algorithmes peuvent être exécutés par un processeur dans une architecture traditionnelle (c'est à dire, avec processeur, mémoire SRAM ...), par un processeur implémenté dans un FPGA, par un circuit dédié, que ce soit un ASIC (Application Specific Integrated Circuit) ou par un circuit au sein d'un dispositif complexe, comme par exemple un processeur *multicore*. Les algorithmes auto-convergeants, quand exécutés par un processeur par exemple, sont exposés aux fautes transitoires (particulièrment aux SEUs) provoquées par exemple par les radiations présentes dans l'environnement, ceci en raison de la sensibilité aux SEUs de ses cellules mémoire (registres, flip flops, caches d'instructions et de données). A titre d'exemple, si les SEUs affectent des données d'entrée ou de sortie de l'algorithme, l'algorithme convergera mais pas au bon

résultat, c'est à dire ces données ne peuvent pas être corrigées par l'algorithme, ce qui empêchera sa convergence vers une valeur correcte.

Les premières campagnes d'injection de fautes, avec l'approche CEU, ont été réalisées sur un processeur LEON3, exécutant l'algorithme d'auto-convergence avec le but d'explorer la robustesse de cet algorithme face aux fautes de type SEU.

Avec le but d'obtenir la pathologie des erreurs qui échappent aux capacités de tolérance aux fautes de l'algorithme d'auto-convergence il a été proposé travailler avec l'algorithme d'auto-convergence au niveau du langage assembler. Des campagnes d'injection des fautes avec l'approche CEU ont été effectuées pour évaluer la sensibilité aux SEUs des variables de l'algorithme et pour identifier les cibles les plus sensibles aux soft-errors.

Des modifications ont été effectuées sur le code C de l'algorithme d'auto-convergence avec le but d'améliorer sa tolérance aux fautes. Un processeur à trois *cores* a été émulé pour implémenter sur le testeur ASTERICS une architecture TMR (Triple Modular Redundancy) exécutant chacun des *cores* l'algorithme d'auto-convergence. Cette technique de tolérance aux fautes a été évaluée par injection de fautes, et les résultats de ces injections ont mis en évidence l'efficacité du TMR implémenté alliée à l'auto-convergence de l'algorithme.

Finalement, nous avons exploré l'impact des fautes injectées, avec l'approche NETFI, émulant le phénomène SEU sur l'algorithme d'auto-convergence implémenté dans une version hardware dans un FPGA. Les résultats des campagnes de test ont confirmé la capacité de fournir des résultats corrects en présence de fautes affectant les cellules mémoire et les données utilisées par l'algorithme.

Parmi les perspectives de ces travaux, peuvent être mentionnés la réalisation des campagnes de test sous radiation pour confronter les résultats d'injection de fautes à des erreurs provoquées dans l'environnement réel. Dans ce contexte, une autre perspective pour ces recherches concerne l'implémentation de l'algorithme d'auto-convergence dans un processeur *multicore* pour évaluer la communication entre les *cores*.

Publications Pendant la Thèse

Journaux

R. Velazco, W. Mansour, F. Pancher, G. Marques-Costa, D. Sohier, A. Bui, "Improving SEU fault tolerance capabilities of a self-converging algorithm", IEEE Transactions on Nuclear Science, vol. 59, pp. 818-823, March 2012.

Conférences et Workshops

R. Velazco, G. Foucard, F. Pancher, W. Mansour, G. Marques-Costa, D. Sohier, A. Bui, "Robustness with respect to SEUs of a self-converging algorithm", 12th Latin American Test Workshop (LATW'11), Porto de Galinhas, Brazil, pp. 1-5, 27-30 March 2011.

G. Marques-Costa, W. Mansour, F. Pancher, R. Velazco, D. Sohier, A. Bui, "Optimization of a self-converging algorithm at assembly level to improve SEU fault-tolerance", 4th Latin American Symposium on Circuits and Systems (LASCAS'13), Cusco, Peru, pp. 1-4, 27 Feb. - 1 March 2013.

W. Mansour, G. Marques-Costa, R. Velazco, "Robustness with respect to SEU of hardware-implemented self-converging algorithm", 25th International Conference on Microelectronics (ICM'13), Beirut, Lebanon, pp. 1-4, 15-18 Dec. 2013.

Bibliographie

[1] A. H. Johnston, "Scaling and technology issues for Soft-Error rates". Proceedings of the 4th Annual Research Conference on Reliability, Stanford University, pp. 567-574, Oct. 2000.

[2] T. P. Ma et al, "Ionizing radiation effects in MOS devices and circuits", Wiley Eds., New York, 1989.

[3] R. Baumann, "Soft Errors in advanced computer systems", IEEE Design and Test of Computers, vol. 22, no. 3, pp. 258-266, May-June 2005.

[4] E. Normand, "Single-Event Effects in avionics", IEEE Transactions on Nuclear Science, vol. 43, no.2, pp. 461-474, April 1996.

[5] M. Radetzki et al, "Methods for fault tolerance in networks-on-chip", Journal ACM Computing Surveys (CSUR), Vol. 46, No. 1, Oct. 2013.

[6] T. Bjerregaard et al, "A survey of research and practices of network-on-chip", ACM Computing Surveys. Vol. 38, No. 1, pp. 1-51, March 2006.

[7] W. Mansour, "Méthodes et outils pour l'analyse tôt dans le flot de conception de la sensibilité aux Soft-Erreurs des applications et des circuits intégrés", PhD thesis, Institut National Polytechnique de Grenoble, Octobre 2012.

[8] J. R. Schwank et al, "Radiation hardness assurance testing of microelectronic devices and integrated circuits: radiations environments, physical mechanisms, and foundations for hardness assurance", IEEE Transaction on Nuclear Science, vol. 60, no. 3, pp. 2074-2100, June 2013.

[9] D. Binder et al, "Satellite anomalies from galactic cosmic rays", IEEE Transaction on Nuclear Science, Vol. 22, No. 6, pp. 2675-2680, Dec. 1975.

[10] J. Ziegler et al, "SER-history, trends and challenges. A guide for designing with memory ICs". Technical report, Cypress Semiconductor, 2005.

[11] J. Wakerly, "Error Detecting Codes, Self-Checking Circuits and Applications", Elsevier North-Holland, New York, NY, USA, 1978.

[12] R. Hamming, "Coding and Information Theory", Prentice-Hall, 1980.

[13] S. Vanstone and P. van Oorschot, "An introduction to error correcting codes with applications", Kluwer, 1989.

[14] W. A. Geisel, "Tutorial on Reed-Solomon error correction coding", Technical Memorandum, NASA, August 1990.

[15] Xuemin Chen, Irving S. Reed, "Error-control coding for data networks", Boston, Kluwer Academic Publishers Norwell, MA, USA 1999.

[16] J. C. Boudenot. L'environnement spatial, collection « Que sais-je ? », Ed. Presses Universitaires de France, 1995.

[17] www.telesat.ca/fre/satellite_transmission5.htm

[18] R.L. Fleischer et al, "Nuclear tracks in solids: principles and applications", University of California Press, January 1975.

[19] E. G. Stassinopoulos et al, "The space radiation environment for electronics", Proceedings of the IEEE, Vol. 76, No. 11, pp. 1423-1442, Nov. 1988.

[20] P. B. Price et al, "Composition of cosmic rays of atomic number 12 to 30", Proceedings of the 11th International Conference on Cosmic Rays, Budapest, Hungary, pp.417-422, 25 Aug - 4 Sep, 1969.

[21] Analyses Professionnelles. Tenue aux Radiations, Tomes 1& 2.

[22] J. L. Barth, "Modeling space radiation environments", IEEE Nuclear and Space Radiation Effects Short Course, Snowmass Village, July 1997.

[23] E. Petersen, "Soft errors due to protons in the radiation belt", IEEE Transaction on Nuclear Science, Vol. 28, No. 6, pp, 3981-3986, Dec 1981.

[24] C. Vial, "Evaluation de la probabilité des aléas logiques induits par les neutrons atmosphérique dans le silicium des SRAM.", PhD thesis, Université Montpellier II, Octobre,1998.

[25] R. Baumann et al, "Boron compounds as a dominant source of alpha particles in semiconductor devices", IEEE International Reliability Physics Symposium, pp. 297-302, April1995.

[26] T. Granlund et al, "SEUs induced by thermal to high-energy neutrons in SRAMs", IEEE Transactions on Nuclear Science, Vol. 53, No. 6, pp. 3798-3802, Dec. 2006.

[27] F. Faure, "Injection de fautes simulant les effets de basculement de bits induits par radiation", PhD thesis, Institut National Polytechnique de Grenoble, Novembre 2005.

[28] E. Normand et al, "Single event upset and charge collection measurements using high energy protons and neutrons", IEEE Transactions on Nuclear Science, Vol. 41, No. 6, Dec. 1994.

[29] JEDEC Standard no. JESD89, "Measurements and reporting of alpha particles and terrestrial cosmic ray-induced soft errors in semiconductor devices", Aug. 2001.

[30] T. R. Oldham et al, "Total ionizing dose effects in MOS oxides and devices". IEEE Transactions on Nuclear Science, Vol. 50, No. 3, pp. 483-499, June 2003.

[31] H. E. Boesch et al, "Charge yield and dose effects in MOS capacitors at 80 K", IEEE Transactions on Nuclear Secience, Vol. 23, No. 6, pp. 1520-1525, Dec. 1976.

[32] T. R. Oldham et al, "Spatial dependence of trapped holes determined from tunneling analysis and measured annealing", IEEE Transactions on Nuclear Science, Vol. 33, No. 6, pp. 1203-1209, Dec. 1986.

[33] F. B. McLean, "A direct tunneling model of charge transfer at the insulator-semicondutor interface in MIS devices", U.S. Gouvernment Report HDL-TR-1975, October 1976.

[34] F. B. McLean et al, "Hole transport and recovery characteristics of SiO_2 gate insulators", IEEE Transactions on Nuclear Science, Vol. 23, No.6, pp. 1506-1512, Dec. 1976.

[35] R. C. Hughes et al, "Hole transport in MOS oxides", IEEE Transactions on Nuclear Science, Vol. 22, No. 6, pp. 2227-2233, Dec. 1975.

[36] S. Duzellier, "Space radiation environment and its effects on spacecraft components and systems, Chapter in Single Event Effects : Anlaysis and Testing", Ed. Cepadues, pp. 221-242, June 2004.

[37] F. Wang et al, "Single event upset : an embedded tutorial", 21st, International Conference on VLSI Design (VLSID), Hyderabad, India, pp. 429-434, Jan. 2008.

[38] S. B. Nicolescu, "Détection d'erreurs transitoires survenant dans des architectures digitales par une approche logicielle : principes et résultats expérimentaux", PhD thesis. Institut National Polytechnique de Grenoble, 2002.

[39] S. Buchner et al, "Pulsed laser validation of recovery mechanism of critical SEEs in artificial network system", Proceeding of 4th European Conference on Radiation and its Effects on Component and Systems (RADECS 97). Cannes, France, pp.110-111, 1997.

[40] F. W. Sexton et al, "Precursor ion damage and angular dependence of single event gate rupture in thin oxides", IEEE Transactions on Nuclear Science, Vol. 45, No. 6, pp. 2509-2518, Dec. 1998.

[41] G. Tel, "Introduction to distributed algorithms", Cambridge University Press, Second Edition, October 2000.

[42] D. Sohier, "Marches aléatoires dans les systémes complexes : exemples du contrôle aérien et des algorithmes distribués à base de marches aléatoires", PhD thesis, École Pratique des Hautes Etudes, 2005.

[43] S. Tixeuil, "Algorithms and theory of computation handbook, chapter self-stabilizing algorithms", Chapman & Hall/CRC Applied algorithms and data structures, Second Edition, CRC Press, Taylor & Francis Group, pp. 26.1-26.45, Nov. 2009.

[44] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control", Magazine Communications of the Association of the Computing Machinery, Vol. 17, No. 11, pp. 643-644, Nov. 1974.

[45] S. Dolev, "Self-stabilization", MIT Press, March 2000.

[46] G. Varghese et al, "The fault span of crash failures", Journal of the ACM, Vol. 47, No. 2, pp. 244-293, March 2000.

[47] C. Johnen et al, "Auto-stabilisation et protocoles réseau", Technique et Science Informatique, Vol. 23, No. 8, pp. 1027-1056, 2004.

[48] J. R. Norris, "Markov chains", Cambridge Series in Statiscal and Probabilistic Mathematics. Cambridge University Press, New York, NY, first edition, 1997.

[49] B. Awerbuch, "Complexity of network synchronization", Jounal of the Association of the Computing Machinery, Vol. 32, no.4, pp. 804-823, Oct. 1985.

[50] K. M. Chandy et al, "Distributed computation on graphs : shortest path algorithms", Magazine Communications of the ACM, Vol. 25, No.11, pp. 833-837, Nov. 1982.

[51] W. D. Tajibnapis, "A correctness proof of a topology information maintenance protocol for a distributed computer network", Magazine Communications of the ACM, Vol. 20, No. 7, pp. 477-485, July 1977.

[52] G. L. Lann, "Distributed systems : towards a formal approach", Proceedings of the International Federation for Information Processing (IFIP) Congress, Toronto, Canada, pp. 155-160, August 1977.

[53] E. Chang et al, "An improved algorithm for decentralized extrema-finding in circular configurations of processes", Magazine Communications of the ACM, Vol. 22, No. 5, pp.281-283, May 1979.

[54] K. M. Chandy et al, "Distributed snapshots : determining global satates of distributed systems", Journal ACM Transactions on Computer Systems (TOCS), Vol. 3, No.1, pp. 63-75, Feb. 1985.

[55] J. Misra, "Detecting termination of distributed computations using markers", Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing, Montreal, Canada, pp. 290-294, August 17-19, 1983.

[56] S. Devismes, "Quelques contributions à la stabilisation instantanée", PhD thesis. L'Université de Picardie Jules Verne, Déc. 2006.

[57] B. Awerbuch et al, "Self-stabilizing end-to-end communication", Journal of High Speed Networks, Vol. 5 No. 4, pp. 365-381, 1996.

[58] B. Awerbuch et al, "Self-stabilization by local checking and correction", Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, San Juan, Porto Rico, pp. 268-277, Oct. 1-4, 1991.

[59] J. Kumagai, "Life of birds [wireless sensor network for bird study]", IEEE Spectrum, Vol. 41, No. 4, pp. 42-49, April 2004.

[60] T. Herman, "Models of self-stabilization and sensor networks", Proceedings of the 5th International Workshop on Distributed Computing, Kolkata, India, pp. 205-214, Dec. 27-30, 2003.

[61] M. Arumugam et al, "Self-stabilizing deterministic TDMA for Sensor Networks", Proceedings of the 2nd International Conference on Distributed Computing and Internet Technology (ICDCIT), Bhubaneswar, India, pp. 69-81, Dec. 22-24, 2005.

[62] D. Bein et al, "A Self-stabilizing directed diffusion protocol for sensor networks", Proceedings of the 33rd International Conference on Parallel Processing Workshops (ICPP), Montreal, Canada, pp. 69-76, Aug. 15-18, 2004.

[63] C. Intanagonwiwat et al, "Directed diffusion: a scalable and robust communication paradigm for sensor network", Proceeding of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom), Boston, USA, pp. 56-67, Aug. 6-11, 2000.

[64] T. Herman et al, "Temporal Partition in Sensor Networks", Proceedings of the 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems, Paris, France, pp. 325-339, Nov. 14-16, 2007.

[65] C. Weyer et al, "Programming wireless sensor networks in a self-stabilization style", Proceedings of the 3rd International Conference on Sensor Technologies and Applications (SENSORCOMM'09), Athens, Greece, pp. 610-616, June 18-23, 2009.

[66] T. Herman et al, "Stabilizing clock synchronization for wireless sensor networks", Best paper of 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems, Dallas, USA, pp. 335-349, Nov. 17-19, 2006.

[67] P. Juang et al, "Energy-efficient computing for wildlife tracking : design tradeoffs and early experiences with ZebraNet", Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), San Jose, USA, pp. 96-107, Oct. 5-9, 2002.

[68] S. Beauquier et al, "Self-stabilizing counting in mobile sensor networks with a base station", Proceedings of the 21st International Symposium on Distributed Computing (DISC), Lemesos, Cyprus, pp. 63-76, Sep. 24-26, 2007.

[69] D. Angluin et al, "Self-stabilizing population protocols", Journal ACM Transactions on Autonomus and Adaptive Systems (TAAS), Vol. 3, No. 4, Nov. 2008.

[70] T. Huang, "An efficient fault-containing self-stabilizing algorithm for the shortest path problem", Journal Distributed Computing, vol. 19, no. 2, pp. 142-161, Oct. 2006.

[71] K. Kakugawa et al, "A self-stabilizing minimal dominating set algorithm with safe convergence", Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS), Rhodes Island, USA, April 25-29, 2006.

[72] F. Avril et al, "A distributed hierarchical clustering algorithm for large-scale dynamic", Proceedings of the 8th ACM Workshop on Performance Monitoring and Measurement of Heterogeneous Wireless an Wired Networks, Barcelona, Spain, pp.197-202, Nov 3-8, 2013.

[73] T. Bernard et al, "Universal adaptive self-stabilizing traversal scheme : random walk and reloaging wave", Journal of Parallel and Distributed Computing, Vol. 73, No. 2, Feb. 2013.

[74] A. Bui et al, "Distributed construction of nested clusters with inter-cluster routing", IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW), Shanghai, China, pp. 673-680, May 16-20, 2011.

[75] A. Bui et al, "Snap-stabilization and PIF in tree networks", Journal Distributed Computing, Vol. 20, No 1, pp. 3-19, July 2007.

[76] S. Rezgui, "Prédiction du taux d'erreurs d'architectures digitales: une méthode et des résultats expérimentaux", PhD thesis, Institut National Polytechnique de Grenoble, 2001.

[77] P. Peronnard, "Méthodes et outils l'évaluation de la sensibilité de circuits intégrés avancés face aux radiations naturelles. PhD thesis, Institut National Polytechnique de Grenoble, 2009.

[78] R. Velazco et al, "Predicting error rate for microprocessor-based digital architectures through CEU (Code Emulating Upsets) injection", IEEE Transaction Nuclear Science, Vol. 47 No. 6, pp. 2405-2411, Dec. 2000.

[79] R. Velazco et al, "Error rate prediction of digital architectures: test methodology and tools", Chapter in Radiation Effects on Embedded Systems, Ed. Springer Netherlands, pp. 233-258, 2007.

[80] R. H. Barned, "The 8051 Family Microcontroller", Prentice-Hall, 1995.

[81] http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.gaisler.com/products/leon2/leon.html

[82] http://fr.wikipedia.org/wiki/Fen%C3%AAtre_de_registres

[83] R. Velazco et al, "THESIC: A testbed suitable for the qualification of integrated circuits devoted to operate in harsh environment", IEEE European Test Workshop, Sitges, Spain, pp. 89-90, May, 1998.

[84] R. Velazco et al. THESIC : A Flexible Plataform for the Functional Validation of Integrated Circuits. IBERCHIP, Mars 2001.

[85] F. Faure et al, "THESIC+: A flexible system for SEE testing", Proceeding of RADECS Workshop, Padova, Italy, pp. 231-234, 2002.

[86] P. Peronnard et al, "Predicting the SEU error rate through fault injection for a complex microprocessor", IEEE International Symposium on Industrial Electronics - ISIE'08, Cambridge, UK, pp. 2288-2292, June 30 - July 2,2008.

[87] http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=13&Itemid=53

[88] E. W. Dijkstra, "A note on two problems in connexion with graphs", Journal Numerische Mathematik, Vol.1, No. 1, pp. 269-271, Dec. 1959.

[89] R. Velazco et al, "Robustness with respect to SEUs of a self-converging algorithm", Procceding of Latin American Test Workshop (LATW 2011), Porto de Galinhas, Brazil, pp. 1-5, March 27-30, 2011.

[90] R. Velazco et al, "Improving SEU fault tolerance capabilities of a self-converging algorithm", IEEE Transaction Nuclear Science, Vol. 59 No. 4, pp. 818-823, Aug. 2012.

[91] G. Marques. C. et al, "Optimization of a self-converging algorithm at assembly level to improve SEU fault-tolerance", IEEE 4th Latin American Symposium on Circuits and Systems (LASCAS), Cusco, Peru, pp. 1-4, Feb. 27 - March 1, 2013.

[92] T. G. Mattson et al, "The 48-core SCC processor : the programmer's view", International Conference for High Performance Computing, Networking, Storage and Analysis, New Orlean, USA, pp. 1-11, Nov 13-19, 2010.

[93] S. Bell et al, "TILE64 processor : a 64-core SoC with mesh interconnect", IEEE International Solid-State Circuits Conference (ISSCC), Digest of Technical Papers, San Francisco, USA, pp. 88-598, Feb. 3-7, 2008.

[94] S. R. Vangal et al, "An 80-Tile sub-100-W TeraFLOPS processor in 65-nm CMOS", IEEE Journal of Solid-State Circuits, Vol. 43, No. 1, pp. 29-41, Jan. 2008.

[95] W. Mansour et al, "An automated SEU fault-injection method and tool for HDL-based designs", IEEE Transactions on Nuclear Science, Vol. 60, No. 4, pp. 2728-2733, Aug. 2013.

[96] W. Mansour et al, "Robustness with respect to SEU of hardware-implemented self-converging algorithm", Proceedings of the 25th International Conference on Microelectronics (ICM), Beirut, Lebanon, pp. 1-4, Dec. 15-18, 2013.

Annexe A. Le tableau T généré aléatoirement

```
n=16

test="{"
m=$((n*n-1))
i=0
while [ $i -le $m ]
	do
		i=$((i+1))
		a=$((RANDOM%16))
		if [ $a -ge 4 ]
			then
				test+="$a, "
			else
				test+="BEAUCOUP, "
		fi
done
a=$((RANDOM%16))
if [ $a -ge 4 ]
	then
		test+="$a}"
	else
		test+="BEAUCOUP}"
fi
	echo $test
```

Annexe B. Exemple d'exécution du programme PCCAS

Ci-dessous figure une exécution du programme, avec la valeur des variables. Le tableau D n'est pas initialisé, et comporte donc des valeurs des variables en début d'exécution. Ces valeurs peuvent être considérées comme résultant d'une corruption de *D* ou de *m* en cours d'exécution. En effet, le nœud 0 est toujours à distance 0 de lui-même, et l'algorithme commence toute itération en mettant D[0] à 0.

Sur chaque ligne de l'exécution est représentée une boucle ***for*** sur *j*. Ainsi, la ligne :

$D[8] = 64192 : \ m = 64128, 10, 10, 10, 10, 10, 10, 8, 8, 8, 8, 4, 4, 4, 4, 4 \rightarrow D[8] = 4 : \text{modification} \ (b = 1)$

Signifie que, au départ la distance de 8 à 0 est estimée à 64192, qu'en allant directement de 8 à 0, on trouve une distance de 64128, qu'en passant par 1, on trouve une distance de 10, en passant par les nœuds 2 à 6, on trouve des distances plus grandes que 10, qu'en passant par 7, on trouve une distance de 8. En passant par les nœuds 8 à 10, on trouve des distances plus grandes que 8, qu'en passant par 11, on trouve une distance de 4, et que, enfin, en passant par les autres nœuds, on trouve une distance supérieure à 4. La distance, telle qu'estimée à cette étape, entre 8 et 0 est donc de 4. Comme cette distance n'est plus la même qu'au départ, l'algorithme n'a pas convergé, et il faudra faire au moins une itération supplémentaire, en fait, il faudra faire au moins deux itération supplémentaires à cause de la variable c et afin d'avoir une meilleure robustesse aux pannes.

Si on considère les valeurs pour *D* au début d'exécution données par :

$$D = [0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 64192,\ 24511,\ 32767,\ 0,\ 4152,\ 0,\ 1,\ 0]$$

Après la première itération :

$D[0]=0$

$D[1]=0: \quad m=9,9,4,4,4,4,4,4,4,4,4,4,4,4,4,4 \rightarrow D[1]=4: \quad \text{modification } (b=1)$

$D[2]=0: \quad m=64128,8,5,5,5,5,5,5,5,5,5,5,5,5,5,5 \rightarrow D[2]=5: \quad \text{modification } (b=1)$

$D[3]=0: \quad m=14,14,10,10,5,5,5,5,5,5,5,5,5,5,5,5 \rightarrow D[3]=5: \quad \text{modification } (b=1)$

$D[4]=0: \quad m=7,7,7,7,6,6,6,6,6,6,6,6,6,6,5,5 \rightarrow D[4]=5: \quad \text{modification } (b=1)$

$D[5]=0: \quad m=9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9 \rightarrow D[5]=9: \quad \text{modification } (b=1)$

$D[6]=0: \quad m=15,9,9,9,9,9,9,8,8,8,8,8,8,8,7,7 \rightarrow D[6]=7: \quad \text{modification } (b=1)$

$D[7]=0: \quad m=4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4 \rightarrow D[7]=4: \quad \text{modification } (b=1)$

$D[8]=64192: \quad m=64128,10,10,10,10,10,10,8,8,8,8,4,4,4,4,4 \rightarrow D[8]=4: \quad \text{modification } (b=1)$

$D[9]=24511: \quad m=6,6,6,6,6,6,6,6,6,6,6,6,6,6,5,5,5 \rightarrow D[9]=5: \quad \text{modification } (b=1)$

$D[10]=32767: \quad m=9,9,9,9,9,9,9,9,9,9,9,9,8,8,8,8,7 \rightarrow D[10]=7: \quad \text{modification } (b=1)$

$D[11]=0: \quad m=7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,6 \rightarrow D[11]=6: \quad \text{modification } (b=1)$

$D[12]=4152: \quad m=10,10,10,10,10,10,10,9,9,9,9,9,9,9,9,9 \rightarrow D[12]=9: \quad \text{modification } (b=1)$

$D[13]=0: \quad m=15,15,15,15,10,10,10,10,10,10,10,10,10,10,7,7 \rightarrow D[13]=7: \quad \text{modification } (b=1)$

$D[14]=1: \quad m=15,15,15,15,9,9,9,9,9,9,9,9,9,9,9,9 \rightarrow D[14]=9: \quad \text{modification } (b=1)$

$D[15]=0: \quad m=6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6 \rightarrow D[15]=6: \quad \text{modification } (b=1)$

Après la deuxième itération :

$D[0]=0$

$D[1]=4: \quad m=9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9 \rightarrow D[1]=9: \quad \text{modification } (b=1)$

$D[2]=5: \quad m=64128,13,10,10,10,10,10,10,10,10,10,10,10,10,10,10 \rightarrow D[2]=10: \quad \text{modification } (b=1)$

$D[3]=5: \quad m=14,14,14,14,10,10,10,10,10,10,10,10,10,10,10,10 \rightarrow D[3]=10: \quad \text{modification } (b=1)$

$D[4]=5: \quad m=7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7 \rightarrow D[4]=7: \quad \text{modification } (b=1)$

$D[5]=9: \quad m=9,9,9,9,9,9,9,9,8,8,8,8,8,8,8,8 \rightarrow D[5]=8: \quad \text{modification } (b=1)$

$D[6]=7: \quad m=15,14,14,14,14,14,14,12,12,12,12,12,12,12,12,12 \rightarrow D[6]=12: \quad \text{modification } (b=1)$

$D[7]=4: \quad m=4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4 \rightarrow D[7]=4: \quad (b=0)$

$D[8]=4: \quad m=64128,15,15,15,15,15,15,8,8,8,8,8,8,8,8,8 \rightarrow D[8]=8: \quad \text{modification } (b=1)$

$D[9]=5: \quad m=6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6 \rightarrow D[9]=6: \quad \text{modification } (b=1)$

$D[10]=7: \quad m=9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9 \rightarrow D[10]=9: \quad \text{modification } (b=1)$

$D[11]=6: \quad m=7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,6 \rightarrow D[11]=7: \quad \text{modification } (b=1)$

$D[12]=9: \quad m=10,10,10,10,10,10,10,9,9,9,9,9,9,9,9,9 \rightarrow D[12]=9: \quad (b=0)$

$D[13]=7: \quad m=15,15,15,15,12,12,12,11,11,11,11,11,11,11,11,11 \rightarrow D[13]=11: \quad \text{modification } (b=1)$

$D[14]=9: \quad m=15,15,15,15,11,11,11,11,11,11,11,11,11,11,11,11 \rightarrow D[14]=11: \quad \text{modification } (b=1)$

$D[15]=6: \quad m=6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6 \rightarrow D[15]=6: \quad (b=0)$

Après la troisième itération :

$D[0]=0$
$D[1]=9: \quad m=9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9 \rightarrow D[1]=9: \quad (b=0)$
$D[2]=10: \quad m=6412\text{8},13,13,13,13,13,13,13,13,13,13,13,13,13,13,13 \rightarrow D[2]=13: \quad \text{modification} \quad (b=1)$
$D[3]=10: \quad m=14,14,14,14,12,12,12,12,12,12,12,12,12,12,12,12 \rightarrow D[3]=12: \quad \text{modification} \quad (b=1)$
$D[4]=7: \quad m=7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7 \rightarrow D[4]=7: \quad (b=1)$
$D[5]=8: \quad m=9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9 \rightarrow D[5]=9: \quad \text{modification} \quad (b=1)$
$D[6]=12: \quad m=15,14,14,14,14,14,14,12,12,12,12,12,12,12,12,12 \rightarrow D[6]=12: \quad (b=0)$
$D[7]=4: \quad m=4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4 \rightarrow D[7]=4: \quad (b=0)$
$D[8]=8: \quad m=6412\text{8},15,15,15,15,15,15,8,8,8,8,8,8,8,8,8 \rightarrow D[8]=8: \quad (b=0)$
$D[9]=6: \quad m=6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6 \rightarrow D[9]=6: \quad (b=0)$
$D[10]=9: \quad m=9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9 \rightarrow D[10]=9: \quad (b=0)$
$D[11]=7: \quad m=7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,6 \rightarrow D[11]=7: \quad (b=0)$
$D[12]=9: \quad m=10,10,10,10,10,10,10,9,9,9,9,9,9,9,9,9 \rightarrow D[12]=9: \quad (b=0)$
$D[13]=11: \quad m=15,15,15,15,12,12,12,11,11,11,11,11,11,11,11,11 \rightarrow D[13]=11: \quad (b=0)$
$D[14]=11: \quad m=15,15,15,15,11,11,11,11,11,11,11,11,11,11,11,11 \rightarrow D[14]=11: \quad (b=0)$
$D[15]=6: \quad m=6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6 \rightarrow D[15]=6: \quad (b=0)$

Après la quatrième itération :

$D[0]=0$
$D[1]=9: \quad m=9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9 \rightarrow D[1]=9: \quad (b=0)$
$D[2]=13: \quad m=6412\text{8},13,13,13,13,13,13,13,13,13,13,13,13,13,13,13 \rightarrow D[2]=13: \quad (b=0)$
$D[3]=12: \quad m=14,14,14,14,12,12,12,12,12,12,12,12,12,12,12,12 \rightarrow D[3]=12: \quad (b=0)$
$D[4]=7: \quad m=7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7 \rightarrow D[4]=7: \quad (b=0)$
$D[5]=9: \quad m=9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9 \rightarrow D[5]=9: \quad (b=0)$
$D[6]=12: \quad m=15,14,14,14,14,14,14,12,12,12,12,12,12,12,12,12 \rightarrow D[6]=12: \quad (b=0)$
$D[7]=4: \quad m=4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4 \rightarrow D[7]=4: \quad (b=0)$
$D[8]=8: \quad m=6412\text{8},15,15,15,15,15,15,8,8,8,8,8,8,8,8,8 \rightarrow D[8]=8: \quad (b=0)$
$D[9]=6: \quad m=6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6 \rightarrow D[9]=6: \quad (b=0)$
$D[10]=9: \quad m=9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9 \rightarrow D[10]=9: \quad (b=0)$
$D[11]=7: \quad m=7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,6 \rightarrow D[11]=7: \quad (b=0)$
$D[12]=9: \quad m=10,10,10,10,10,10,10,9,9,9,9,9,9,9,9,9 \rightarrow D[12]=9: \quad (b=0)$
$D[13]=11: \quad m=15,15,15,15,12,12,12,11,11,11,11,11,11,11,11,11 \rightarrow D[13]=11: \quad (b=0)$
$D[14]=11: \quad m=15,15,15,15,11,11,11,11,11,11,11,11,11,11,11,11 \rightarrow D[14]=11: \quad (b=0)$
$D[15]=6: \quad m=6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6 \rightarrow D[15]=6: \quad (b=0)$

Après la quatrième itération :

$D[0] = 0$
$D[1] = 9: \quad m = 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9 \rightarrow D[1] = 9: \quad (b = 0)$
$D[2] = 13: \quad m = 64128, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13 \rightarrow D[2] = 13: \quad (b = 0)$
$D[3] = 12: \quad m = 14, 14, 14, 14, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12 \rightarrow D[3] = 12: \quad (b = 0)$
$D[4] = 7: \quad m = 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7 \rightarrow D[4] = 7: \quad (b = 0)$
$D[5] = 9: \quad m = 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9 \rightarrow D[5] = 9: \quad (b = 0)$
$D[6] = 12: \quad m = 15, 14, 14, 14, 14, 14, 14, 12, 12, 12, 12, 12, 12, 12, 12, 12 \rightarrow D[6] = 12: \quad (b = 0)$
$D[7] = 4: \quad m = 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 \rightarrow D[7] = 4: \quad (b = 0)$
$D[8] = 8: \quad m = 64128, 15, 15, 15, 15, 15, 15, 8, 8, 8, 8, 8, 8, 8, 8, 8 \rightarrow D[8] = 8: \quad (b = 0)$
$D[9] = 6: \quad m = 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6 \rightarrow D[9] = 6: \quad (b = 0)$
$D[10] = 9: \quad m = 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9 \rightarrow D[10] = 9: \quad (b = 0)$
$D[11] = 7: \quad m = 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 6 \rightarrow D[11] = 7: \quad (b = 0)$
$D[12] = 9: \quad m = 10, 10, 10, 10, 10, 10, 10, 9, 9, 9, 9, 9, 9, 9, 9, 9 \rightarrow D[12] = 9: \quad (b = 0)$
$D[13] = 11: \quad m = 15, 15, 15, 15, 12, 12, 12, 11, 11, 11, 11, 11, 11, 11, 11, 11 \rightarrow D[13] = 11: \quad (b = 0)$
$D[14] = 11: \quad m = 15, 15, 15, 15, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11 \rightarrow D[14] = 11: \quad (b = 0)$
$D[15] = 6: \quad m = 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6 \rightarrow D[15] = 6: \quad (b = 0)$

A la fin de l'exécution les valeurs attendues pour D sont :

$$D = [0,\ 9,\ 13,\ 12,\ 7,\ 9,\ 12,\ 4,\ 8,\ 6,\ 9,\ 7,\ 9,\ 11,\ 11,\ 6]$$

Annexe C. Code d'injection de SEU dans les registres de la fenêtre de registres : routine ASM d'interruption

```
inject:
           mov %psr, %l0
           set 0x40000000, %l3
           ld [%l3], %g6          ! target
           ld [%l3 + 4], %g7      ! mask

           set inject_l0, %g5
           restore
           jmpl %g5 + %g6, %g0
           nop

inject_l0:
           ba exit_inject
           nop
           xor %l0, %g7, %l0
inject_l1:
           ba exit_inject
           nop
           xor %l1, %g7, %l1
inject_l2:
           ba exit_inject
           nop
           xor %l2, %g7, %l2
inject_l3:
           ba exit_inject
           nop
           xor %l3, %g7, %l3
inject_l4:
           ba exit_inject
           nop
           xor %l4, %g7, %l4
inject_l5:
           ba exit_inject
           nop
           xor %l5, %g7, %l5
inject_l6:
           ba exit_inject
           nop
           xor %l6, %g7, %l6
inject_l7:
           ba exit_inject
           nop
           xor %l7, %g7, %l7
```

```
inject_o0:
            ba exit_inject
            nop
            xor %o0, %g7, %o0
inject_o1:
            ba exit_inject
            nop
            xor %o1, %g7, %o1
inject_o2:
            ba exit_inject
            nop
            xor %o2, %g7, %o2
inject_o3:
            ba exit_inject
            nop
            xor %o3, %g7, %o3
inject_o4:
            ba exit_inject
            nop
            xor %o4, %g7, %o4
inject_o5:
            ba exit_inject
            nop
            xor %o5, %g7, %o5
inject_o7:
            ba exit_inject
            nop
            xor %o7, %g7, %o7
inject_i0:
            ba exit_inject
            nop
            xor %i0, %g7, %i0
inject_i1:
            ba exit_inject
            nop
            xor %i1, %g7, %i1
inject_i2:
            ba exit_inject
            nop
            xor %i2, %g7, %i2
inject_i3:
            ba exit_inject
            nop
            xor %i3, %g7, %i3
inject_i4:
            ba exit_inject
            nop
            xor %i4, %g7, %i4
inject_i5:
            ba exit_inject
            nop
            xor %i5, %g7, %i5
inject_i7:
            ba exit_inject
            nop
            xor %i7, %g7, %i7
```

```
inject_g1:
           ba exit_inject
           nop
           xor %g1, %g7, %g1
inject_g2:
           ba exit_inject
           nop
           xor %g2, %g7, %g2
inject_g3:
           ba exit_inject
           nop
           xor %g3, %g7, %g3

exit_inject:
           save                    ! int handler window

           mov %l0, %psr
           nop
           jmpl %l1, %g0           ! return to program
           rett %l2
```

Annexe D. Code d'injection de SEU dans le compteur de programme (PC) : routine ASM d'interruption

```
inject:
            mov %psr, %l0
            set 0x40000000, %l3
            ld [%l3], %g6           ! target
            ld [%l3 + 4], %g7       ! mask

            xor %l1, %g7, %l1
            restore
            ba exit_inject
            nop

exit_inject:
            save                    ! int handler window

            mov %l0, %psr
            nop
            jmpl %l1, %g0           ! return to program
            rett %l2
```

Annexe E. Code d'injection de SEU dans le next compteur de programme (nPC): routine ASM d'interruption

```
inject:
           mov %psr, %l0
           set 0x40000000, %l3
           ld [%l3], %g6          ! target
           ld [%l3 + 4], %g7      ! mask

           xor %l2, %g7, %l2
           restore
           ba exit_inject
           nop


exit_inject:
           save                   ! int handler window

           mov %l0, %psr
           nop
           jmpl %l1, %g0          ! return to program
           rett %l2
```

Annexe F. Code d'injection de SEU dans la cache d'instructions : routine ASM d'interruption

```
inject:
            mov %psr, %l0
            set 0x40000000, %l3
            ld [%l3], %g6           ! target
            ld [%l3 + 4], %g7       ! mask


            lda [%g6]0x0a,%g5
            xor %g5,%g7,%g5
            sta %g5,[%g6]0x0a
            restore
            ba exit_inject
            nop

exit_inject:
            save                    ! int handler window

            mov %l0, %psr
            nop
            jmpl %l1, %g0           ! return to program
            rett %l2
```

Annexe G. Code d'injection de SEU dans la cache de données : routine ASM d'interruption

```
inject:
            mov %psr, %l0
            set 0x40000000, %l3
            ld [%l3], %g6           ! target
            ld [%l3 + 4], %g7       ! mask

            lda [%g6]0x0f,%g5
            xor %g5,%g7,%g5
            sta %g5,[%g6]0x0f
            restore
            ba exit_inject
            nop

exit_inject:
            save                    ! int handler window

            mov %l0, %psr
            nop
            jmpl %l1, %g0           ! return to program
            rett %l2
```

Annexe H. L'algorithme d'auto-convergence en langage assembleur

```
main:

            set 0x0, %l3
            set 0x1, %l4
            set 0x1, %l5
            set 0x1, %l6
            set 0x0, %l7

whileloop:

            mov %l4,%l5
            set 0x0, %l4
            st %g0, [%l2]
            set 0x1,%l6

foriloop:
            set 0xFA80,%l3
            set 0x00, %l7



forjloop:

            sll %l6,4,%i0
            add %i0,%l7,%i1
            sll %i1,2,%i2
            ld [%l1+%i2],%i3
            sll %l7,2,%i4
            ld [%l2 + %i4], %i5
            add %i5,%i3,%i7
            cmp %i7,%l3
            bl setm
            nop
            ba endj
            nop

setm:
            mov %i7,%l3

endj:

            inc %l7
```

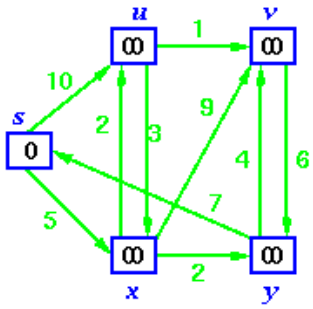
```
            cmp %l7,16
            bl forjloop
            nop

            sll %l6,2,%o1
            ld [%l2 + %o1],%o2
            cmp %l3,%o2
            be endi
            nop
            set 0x1,%l4

endi:

            st %l3,[%l2 + %o1]
            inc %l6
            cmp %l6,16
            bl foriloop
            nop

            or %l4,%l5,%o3
            cmp %g0,%o3
            bne whileloop
            nop
            nop


            nop
            !save
            !save



__exit:
            ! running = 0
            set 0x80000100, %g1
            ld [%g1], %g2
            and %g2, ~4, %g2
            st %g2, [%g1 + 4]

__end:
            ba __end
            nop
```

Annexe I. Un exemple du fonctionnement d'algorithme de Dijkstra

Un exemple du fonctionnement d'algorithme de Dijkstra est donné dans ce qui suit. Cet exemple recherche le plus court chemin pour aller de *s* à *v*.

1.1 Initialement toutes les noeuds on un coût infini sauf ***s*** (la racine de la recherche) qui a une valeur *0*:

Sommets	**s**	**u**	**v**	**x**	**y**
estimations	0	∞	∞	∞	∞
précédents	-	-	-	-	-

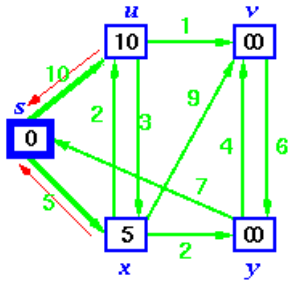


2.1 Sélectionner ***s*** (sommet ouvert d'estimation minimale)

2.2 Fermer ***s***

2.3 Recalculer les estimations de *u* et *x*

Sommets	**s**	**u**	**v**	**x**	**y**
estimations	**0**	10	∞	5	∞
précédents	**s**	s	-	s	-

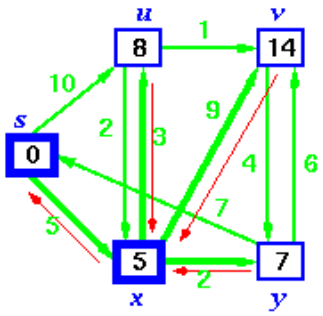


3.1 Sélectionner ***x*** (sommet ouvert d'estimation minimale)

3.2 Fermer ***x***

3.3 Recalculer les estimations de *u*, *v* e *y*

Sommets	s	u	v	x	y
estimations	**0**	8	14	5	7
précédents	**s**	x	x	s	x

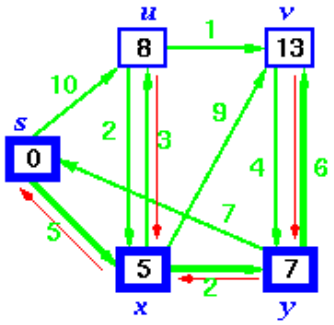


4.1 Sélectionner **y** (sommet ouvert d'estimation minimale)

4.2 Fermer **y**

4.3 Recalculer les estimations de v

Sommets	s	u	v	x	y
estimations	**0**	8	13	5	7
précédents	**s**	x	y	s	x

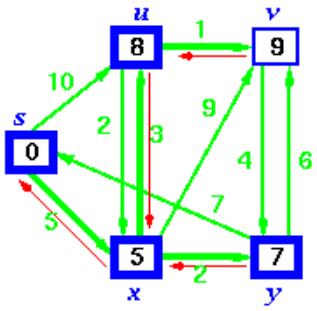


5.1 Sélectionner ***u*** (sommet ouvert d'estimation minimale)

5.2 Fermer ***u***

5.3 Recalculer l'estimation de *v*

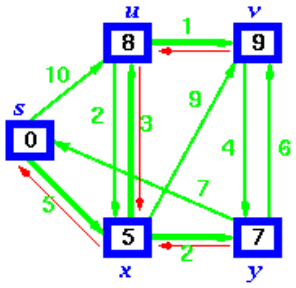
Sommets	s	u	v	x	y
estimations	**0**	**8**	9	**5**	**7**
précédents	**s**	**x**	u	**s**	**x**



6.1 Sélectionner ***v*** (sommet ouvert d'estimation minimale)

6.2 Fermer ***v***

Sommets	**s**	**u**	**v**	**x**	**y**
estimations	**0**	**8**	**9**	**5**	**7**
précédents	**s**	**x**	**u**	**s**	**x**



Lorsque tous les sommets ont été fermées, les valeurs obtenues seront les coûts minimaux des chemins qui partent du sommet pris comme racine de recherche à tous les autres sommets du graphe. Le chemin lui-même est obtenu à partir des sommets précédents. Alors, le plus court chemin allant de *s* à *v* est: $s \rightarrow x \rightarrow u \rightarrow v$, le coût minimum de ce chemin est 9.

Annexe J. Les codes Verilog de l'Algorithme d'auto-convergence

Les modules de l'algorithme d'auto-convergence implémenté dans un FPGA Virtex IV sont donnés dans ce qui suit.

```
`timescale 1ns / 1ps

module node(
   input [11:0] T,
   input [11:0] D,
   input clk_i,
   output reg [11:0] res,
   input rst_n
   );

        always@(posedge clk_i)
        begin
        if(!rst_n)
                res <= 0;
        else res <= T + D;
        end

endmodule
**********************************************************************


`timescale 1ns / 1ps

module cmp2(
   input [11:0] res1,
   input [11:0] res2,
   output reg [11:0] min,
         input clk_i,
         input rst_n
   );

        always@(posedge clk_i)
        begin
        if(!rst_n)
                min <= 0;
        else
        begin
                if(res1 <= res2) min <= res1[11:0];
                else min <= res2[11:0];
        end
```

```
        end

endmodule
**********************************************************************

`timescale 1ns / 1ps

module col2(
    input [23:0] T,
    input [23:0] D,
    output [11:0] min,
    input clk_i,
    input rst_n
    );

        wire [23:0] res;

node U1(
    .T(T[11:0]),
    .D(D[11:0]),
    .clk_i(clk_i),
    .res(res[11:0]),
    .rst_n(rst_n)
        );

node U2(
    .T(T[23:12]),
    .D(D[23:12]),
    .clk_i(clk_i),
    .res(res[23:12]),
    .rst_n(rst_n)
        );

cmp2 UUT(
    .res1(res[11:0]),
    .res2(res[23:12]),
    .min(min),
        .clk_i(clk_i),
        .rst_n(rst_n)
    );

endmodule

**********************************************************************

`timescale 1ns / 1ps

module col4(
    input [47:0] T,
    input [47:0] D,
    output [11:0] min,
    input clk_i,
    input rst_n
    );
```

```
        wire [23:0] res;

        col2 U1(
    .T(T[23:0]),
    .D(D[23:0]),
    .min(res[11:0]),
    .clk_i(clk_i),
    .rst_n(rst_n)
    );

         col2 U2(
    .T(T[47:24]),
    .D(D[47:24]),
    .min(res[23:12]),
    .clk_i(clk_i),
    .rst_n(rst_n)
    );

         cmp2 UUT(
    .res1(res[11:0]),
    .res2(res[23:12]),
    .min(min),
         .clk_i(clk_i),
         .rst_n(rst_n)
    );

endmodule
**********************************************************************

`timescale 1ns / 1ps

module col8(
    input [95:0] T,
    input [95:0] D,
    input clk_i,
    input rst_n,
    output [11:0] min
    );

        wire [23:0] res;

        col4 U0(
    .T(T[47:0]),
    .D(D[47:0]),
    .min(res[11:0]),
    .clk_i(clk_i),
    .rst_n(rst_n)
         );

         col4 U1(
    .T(T[95:48]),
    .D(D[95:48]),
    .min(res[23:12]),
    .clk_i(clk_i),
    .rst_n(rst_n)
```

```
        );

        cmp2 UUT(
    .res1(res[11:0]),
    .res2(res[23:12]),
    .min(min),
        .clk_i(clk_i),
        .rst_n(rst_n)
    );

endmodule
**********************************************************************

`timescale 1ns / 1ps

module col16(
    input [191:0] T,
    input [191:0] D,
    input clk_i,
    input rst_n,
    output [11:0] min
    );

        wire [23:0] res;
        col8 U0(
    .T(T[95:0]),
    .D(D[95:0]),
    .min(res[11:0]),
    .clk_i(clk_i),
    .rst_n(rst_n)
        );

        col8 U1(
    .T(T[191:96]),
    .D(D[191:96]),
    .min(res[23:12]),
    .clk_i(clk_i),
    .rst_n(rst_n)
        );

        cmp2 UUT(
    .res1(res[11:0]),
    .res2(res[23:12]),
    .min(min),
        .clk_i(clk_i),
        .rst_n(rst_n)
    );

endmodule
**********************************************************************

`timescale 1ns / 1ps

module col32(
```

```
   input [383:0] T,
   input [383:0] D,
   input clk_i,
   input rst_n,
   output [11:0] min,
        output rstn_o
      );

        wire [23:0] res;

        col16 U0(
   .T(T[191:0]),
   .D(D[191:0]),
   .min(res[11:0]),
   .clk_i(clk_i),
   .rst_n(rst_n));

         col16 U1(
   .T(T[383:192]),
   .D(D[383:192]),
   .min(res[23:12]),
   .clk_i(clk_i),
   .rst_n(rst_n));

         cmp2 UUT(
   .res1(res[11:0]),
   .res2(res[23:12]),
   .min(min),
         .clk_i(clk_i),
         .rst_n(rst_n)
   );
FD rst_o(.D(rst_n),.Q(rstn_o),.C(clk_i);

endmodule
**********************************************************************

`timescale 1ns / 1ps

module net16x16(
   input clk_i,
   output [191:0] Do,
   input rst_n
   );
parameter n = 16;
reg [191:0] D;
wire [191:0] T0;
wire [191:0] T1;
wire [191:0] T2;
wire [191:0] T3;
wire [191:0] T4;
wire [191:0] T5;
wire [191:0] T6;
wire [191:0] T7;
wire [191:0] T8;
wire [191:0] T9;
```

```
wire [191:0] T10;
wire [191:0] T11;
wire [191:0] T12;
wire [191:0] T13;
wire [191:0] T14;
wire [191:0] T15;
reg [31:0] counter;
reg [3:0] i;
reg [15:0] rst;


assign T0 =
{12'd9,12'd6,12'd13,12'd6,12'd13,12'd2688,12'd2688,12'd2688,12'd11,12'd7,12'd2688,12'd8,12'd8,12'
d2688,12'd5,12'd2688};
assign T1 =
{12'd9,12'd2688,12'd4,12'd14,12'd4,12'd7,12'd2688,12'd2688,12'd2688,12'd13,12'd2688,12'd5,12'd7,
12'd2688,12'd5,12'd5};
assign T2 =
{12'd2688,12'd4,12'd5,12'd9,12'd2688,12'd5,12'd7,12'd15,12'd15,12'd12,12'd6,12'd2688,12'd4,12'd5,
12'd2688,12'd7};
assign T3 =
{12'd14,12'd2688,12'd5,12'd15,12'd5,12'd15,12'd11,12'd13,12'd12,12'd2688,12'd2688,12'd2688,12'd1
1,12'd11,12'd10,12'd14};
assign T4 =
{12'd7,12'd7,12'd5,12'd6,12'd6,12'd2688,12'd14,12'd2688,12'd5,12'd2688,12'd6,12'd2688,12'd5,12'd2
688,12'd4,12'd2688};
assign T5 =
{12'd9,12'd2688,12'd8,12'd5,12'd2688,12'd13,12'd2688,12'd10,12'd4,12'd7,12'd9,12'd2688,12'd14,12'
d10,12'd2688,12'd9};
assign T6 =
{12'd15,12'd5,12'd2688,12'd2688,12'd2688,12'd2688,12'd14,12'd8,12'd2688,12'd2688,12'd12,12'd15,
12'd2688,12'd2688,12'd6,12'd11};
assign T7 =
{12'd4,12'd11,12'd7,12'd8,12'd14,12'd10,12'd2688,12'd2688,12'd2688,12'd2688,12'd2688,12'd13,12'd
11,12'd5,12'd5,12'd14};
assign T8 =
{12'd2688,12'd6,12'd8,12'd9,12'd8,12'd9,12'd2688,12'd4,12'd2688,12'd2688,12'd14,12'd4,12'd2688,1
2'd7,12'd2688,12'd10};
assign T9 =
{12'd6,12'd2688,12'd8,12'd15,12'd6,12'd13,12'd14,12'd2688,12'd9,12'd6,12'd2688,12'd11,12'd7,12'd5
,12'd12,12'd5};
assign T10 =
{12'd9,12'd2688,12'd11,12'd2688,12'd6,12'd5,12'd13,12'd2688,12'd12,12'd12,12'd5,12'd8,12'd2688,1
2'd14,12'd11,12'd7};
assign T11 =
{12'd7,12'd2688,12'd4,12'd4,12'd2688,12'd15,12'd2688,12'd10,12'd8,12'd2688,12'd6,12'd2688,12'd11
,12'd2688,12'd15,12'd6};
assign T12 =
{12'd10,12'd2688,12'd5,12'd2688,12'd2688,12'd11,12'd10,12'd5,12'd6,12'd8,12'd8,12'd2688,12'd2688
,12'd13,12'd13,12'd10};
assign T13 =
{12'd15,12'd2688,12'd2688,12'd14,12'd5,12'd7,12'd15,12'd7,12'd2688,12'd12,12'd2688,12'd15,12'd26
88,12'd2688,12'd6,12'd9};
```

```
assign T14 =
{12'd15,12'd14,12'd14,12'd2688,12'd4,12'd2688,12'd2688,12'd9,12'd10,12'd13,12'd2688,12'd13,12'd2
688,12'd13,12'd14,12'd11};
assign T15 =
{12'd6,12'd8,12'd11,12'd4,12'd11,12'd14,12'd4,12'd2688,12'd4,12'd11,12'd12,12'd6,12'd13,12'd15,12'
d8,12'd6};

always@(posedge clk_i)
begin
D[191:180] <= 0;
        if(!rst_n)
        begin
                D <= 0;
                counter <= 0;
                i <= 1;
                rst <= 0;
        end
        else begin
                rst[n-i] <= 1;
                counter <= counter + 1;
                if(counter == 5)
                begin
                //      D[((12*(n-i))-1):((12*(n-i))-12)] <= Do[((12*(n-i))-1):((12*(n-i))-12)];
                        counter <= 0;
                        rst[n-i-1] <= 0;
                        i <= i + 1;
                end
        end

end


col16 U0(
   .T(T0),
   .D(D),
   .clk_i(clk_i),
   .rst_n(rst_n),
   .min(Do[191:180])
   );

col16 U1(
  .T(T1),
  .D(D),
  .clk_i(clk_i),
  .rst_n(rst[14]),
  .min(Do[179:168])
   );

col16 U2(
  .T(T2),
  .D(D),
  .clk_i(clk_i),
  .rst_n(rst[13]),
  .min(Do[167:156])
   );
```

```
col16 U3(
  .T(T3),
  .D(D),
  .clk_i(clk_i),
  .rst_n(rst[12]),
  .min(Do[155:144])
   );

col16 U4(
  .T(T4),
  .D(D),
  .clk_i(clk_i),
  .rst_n(rst[11]),
  .min(Do[143:132])
   );

col16 U5(
  .T(T5),
  .D(D),
  .clk_i(clk_i),
  .rst_n(rst[10]),
  .min(Do[131:120])
   );

col16 U6(
  .T(T6),
  .D(D),
  .clk_i(clk_i),
  .rst_n(rst[9]),
  .min(Do[119:108])
   );

col16 U7(
  .T(T7),
  .D(D),
  .clk_i(clk_i),
  .rst_n(rst[8]),
  .min(Do[107:96])
   );

col16 U8(
  .T(T8),
  .D(D),
  .clk_i(clk_i),
  .rst_n(rst[7]),
  .min(Do[95:84])
   );

col16 U9(
  .T(T9),
  .D(D),
  .clk_i(clk_i),
  .rst_n(rst[6]),
  .min(Do[83:72])
```

```
    );

col16 U10(
   .T(T10),
   .D(D),
   .clk_i(clk_i),
   .rst_n(rst[5]),
   .min(Do[71:60])
    );

col16 U11(
   .T(T11),
   .D(D),
   .clk_i(clk_i),
   .rst_n(rst[4]),
   .min(Do[59:48])
    );

col16 U12(
   .T(T12),
   .D(D),
   .clk_i(clk_i),
   .rst_n(rst[3]),
   .min(Do[47:36])
    );

col16 U13(
   .T(T13),
   .D(D),
   .clk_i(clk_i),
   .rst_n(rst[2]),
   .min(Do[35:24])
    );

col16 U14(
   .T(T14),
   .D(D),
   .clk_i(clk_i),
   .rst_n(rst[1]),
   .min(Do[23:12])
    );

col16 U15(
   .T(T15),
   .D(D),
   .clk_i(clk_i),
   .rst_n(rst[0]),
   .min(Do[11:0])
    );

endmodule
```

Annexe L. Les modules (*flip-flops*) de Xilinx modifiées

```
module FD_mod (
input inj,
input D,
output Q,
input C);
parameter INIT = 1'b0;
wire Din;
assign Din = (inj) ? !D : D;
FD uut_FDC (.Q(Q), .C(C), .D(Din));
endmodule
**********************************************************************************

module FDC_mod (inj,Q, C, CLR, D);
parameter INIT = 1'b0;
output Q;
input C, CLR, D, inj;
wire Din;
FDC uut(.Q(Q),.C(C),.CLR(CLR),.D(Din));
assign Din = (inj) ? !D : D;
endmodule
**********************************************************************************

module FDCE_mod (inj, Q, C, CE, CLR, D);
parameter INIT = 1'b0;
output Q;
input inj, C, CE, CLR, D;
wire Q;
wire q_out;
wire Din;
FDCE uut(.Q(Q),.C(C),.CE(CE||inj),.CLR(CLR),.D(Din));
assign Din = (inj && !CE) ? !Q : (inj && CE) ? !D : (CE && !inj) ? D : D;
endmodule
**********************************************************************************

module FDCP_mod (inj, Q, C, CLR, D, PRE);
parameter INIT = 1'b0;
output Q;
input inj, C, CLR, D, PRE;
wire Q_n;
wire Din;
wire En;
assign Q_n = Q;
assign Din = (!inj) ? D : !D ;
FDCP uut_FDE(.CLR(CLR),.PRE(PRE), .Q(Q), .D(Din), .C(C) );
```

```
endmodule
**********************************************************************************

module FDE_mod (
input inj,
output Q,
input D,
input C,
input CE);
wire Din;
FDE uut (.Q(Q),.D(Din),.C(C),.CE(CE||inj));
assign Din = (inj && !CE) ? !Q : (inj && CE) ? !D : (CE && !inj) ? D : Q;
endmodule
**********************************************************************************

module FDP_mod (inj,Q, C, D, PRE);
parameter INIT = 1'b0;
output Q;
wire Din;
input inj, C, D, PRE;
FDP uut(.C(C),.D(Din),.PRE(PRE),.Q(Q));
assign Din = (inj) ? !D : D;
endmodule
**********************************************************************************

module FDPE_mod (inj,Q, C, CE, D, PRE);
parameter INIT = 1'b0;
output Q;
input inj, C, CE, D, PRE;
wire Din;
FDPE uut (.Q(Q),.C(C),.CE(CE),.D(Din),.PRE(PRE));
assign Din = (inj && !CE) ? !Q : (inj && CE) ? !D : (CE && !inj) ? D : Q;
endmodule
**********************************************************************************

module FDR_mod (inj, Q, C, D, R);
parameter INIT = 1'b0;
output Q;
input inj, C, D, R;
wire Q;
reg q_out;
assign Q = q_out;
always @(posedge C )
begin
if(!inj)
if (R)
q_out <= 0;
else
q_out <= D;
else
if (R)
q_out <= 1;
else
q_out <= !D;
end
```

```
endmodule
**********************************************************************************

module FDRE_mod (inj, Q, C, CE, D, R);
parameter INIT = 1'b0;
output Q;
input C, CE, D, R, inj;
wire Q;

reg q_out;
assign Q = q_out;
always @(posedge C )
begin
if (R && !inj)
q_out <= 0;
else if (R && inj) q_out <= 1;
else if (CE && !inj)
q_out <= D;
else if(CE && inj) q_out <= !D;
else if (!CE && inj) q_out <= !q_out;
end
endmodule
**********************************************************************************

module FDRSE_mod (inj, Q, C, CE, D, R, S);
parameter INIT = 1'b0;
output Q;
input inj, C, CE, D, R, S;
wire Q;
reg q_out;
assign Q = q_out;
always @(posedge C )
begin
if(!inj)
if (R)
q_out <= 0;
else if (S)
q_out <= 1;
else if (CE)
q_out <= D;
else
if (R) q_out <= 1;
else if (S) q_out <= 0;
else if (CE) q_out <= !D;
else if(!CE) q_out <= !q_out;
end
endmodule
**********************************************************************************

module FDS_mod (inj, Q, C, D, S);
parameter INIT = 1'b1;
output Q;
input inj, C, D, S;
wire Q;
reg q_out;
```

```
assign Q = q_out;
always @(posedge C )
begin
if(!inj)
if (S)
q_out <= 1;
else
q_out <= D;
else
if(S)
q_out <= 0;
else
q_out <= !D;
end
endmodule
```

TITRE
Étude de la fiabilité des algorithmes self-convergeants face aux soft-erreurs

RESUME
Cette thèse est consacrée à l'étude de la robustesse/sensibilité d'un algorithme auto-convergeant face aux SEU's. Ces phénomènes appelés aussi *bit-flips* qui se traduit par le basculement intempestif du contenu d'un élément mémoire comme conséquence de l'ionisation produite par le passage d'une particule chargée avec le matériel. Cette étude pourra avoir un impact important vu la conjoncture de miniaturisation qui permettra bientôt de disposer de circuits avec des centaines à des milliers de cœurs de traitement sur une seule puce, pour cela il faudra faire les cœurs communiquer de manière efficace et robustes. Dans ce contexte les algorithme dits auto-convergeants peuvent être utilis afin que la communication entre les cœurs soit fiable et sans intervention extérieure.
Une étude par injection de fautes de la robustesse de l'algorithme étudié a été effectuée, cet algorithme a été initialement exécuté par un processeur LEON3 implémenté dans un FPGA embarqué dans une plateforme de test spécifique. Les campagnes préliminaires d'injection de fautes issus d'une méthode de l'état de l'art appelée CEU (Code Emulated Upset) ont mis en évidence une certaine sensibilité aux SEUs de l'algorithme. Pour y faire face des modifications du logiciel ont été effectuées et des techniques de tolérance aux fautes ont été implémentés au niveau logiciel dans le programme implémentant l'algorithme. Des expériences d'injection de fautes ont été effectués pour mettre en évidence la robustesse face aux SEUs et ses potentiels « Tallons d'Achille » de l'algorithme modifié. L'impact des SEUs a été aussi exploré sur l'algorithme auto-convergeant implémenté dans une version hardware dans un FPGA. L'évaluation de cette méthodologie a été effectuée par des expériences d'injection de fautes au niveau RTL du circuit. Ces résultats obtenus avec cette méthode ont montré une amélioration significative de la robustesse de l'algorithme en comparaison avec sa version logicielle.

MOTS CLEFS
Environnement spatial, radiations ionisantes, événements singuliers, injection de fautes, tolérance aux fautes, algorithmes tolérants aux fautes, code HDL, architectures à base de processeurs

TITLE
Study of reliability of self-convergent algorithms with respect to soft errors

ABSTRACT
This thesis is devoted to the study of the robustness/sensitivity of a self-converging algorithm with respect to SEU's. These phenomenon also called bit-flips which may modify the content of memory elements as the result of the silicon ionization resulting from the impact of a charged particles. This study may have a significant impact given the conditions of miniaturization that will soon have circuits with hundreds to thousands of processing cores on a single chip, this will require make the cores communicate effectively and robust manner. In this context the so-called self-converging algorithm can be used to ensure that communication between cores is reliable and without external intervention.
A fault injection study of the robustness of the algorithm was performed, this algorithm was initially executed by a processor LEON3 implemented in the FPGA embedded in a specific platform test. Preliminary fault injection from a method the state of the art called CEU showed some sensitivity to SEUs of algorithm. To cope with the software changes were made and techniques for fault tolerance have been implemented in software in the program implementing the self-converging algorithm. The fault injection experiments were made to demonstrate the robustness to SEU's and potential problems of the modified algorithm. The impact of SEUs was explored on a hardware-implemented self-converging algorithm in a FPGA. The evaluation of this method was performed by fault injection at RTL level circuit. These results obtained with this method have shown a significant improvement of the robustness of the algorithm in comparison with its software version.

KEYWORDS
Space environment, ionizing radiation, single event effects, fault-injection, fault tolerance, fault-tolerant algorithms, HDL code, processor-based architectures

INTITULE ET ADRESSE DU LABORATOIRE
Laboratoire TIMA, 46 avenue Félix Viallet, 38031 Grenoble, France.

ISBN :